FaaS-Profiler: Serverless Tracing and Profiling

Bachelor Thesis

Malte Wächter

October 05, 2022

Advisors: Prof. Dr. Torsten Hoefler, Marcin Copik

Department of Computer Science, ETH Zürich

Abstract

Serverless computing or Function-as-a-Service is an emerging and promising cloud execution model. The critical difference between Function-as-a-Service versus a traditional cloud execution method is that applications are broken down into smaller stateless functions that interact with each other driven by results, and the executing infrastructure is abstracted away for ease of use. In this process, the ability to collect performance metrics is hampered. We present the profiling and tracing framework FaaS-Profiler. The framework generalizes executions in a serverless environment. It is designed with the goal of instrumenting serverless functions without much effort, taking measurements to collect metrics, and tracing functions end to end. We evaluate our design with a concrete implementation in Python on Amazon Web Services and Google Cloud Platform.

Contents

Chapter 1

Introduction

Serverless computing or Function-as-a-Service (FaaS) is a new emerging fully-managed cloud computing execution model. In contrast to traditional cloud computing paradigms, FaaS platforms divide applications into independent stateless functions that various events can trigger. The goal of the model is to focus the user on the code without worrying about the underlying infrastructure, which the platform provider entirely manages. All resources required for function execution are allocated on-the-fly by the cloud platform. Serverless computing thus offers the advantage of simple software deployment, high scalability on demand, and concurrency. In addition, the pay-as-you-go method of billing is used, so that you only have to pay for resources that were actually used.

The complete abstraction of the underlying infrastructure has the disadvantage that details about the structure remain hidden from the user and possible code optimizations are more challenging to implement and measure their impact. Furthermore, serverless workflows are built from the interaction of different functions, making constructing a causal dependency between the tasks and collecting performance values across function boundaries difficult.

In this thesis, we introduce FaaS-Profiler. FaaS-Profiler is a profiling framework for serverless functions, with the design goal of collecting metrics independent of programming language and cloud provider. The framework is open and modular, allowing arbitrary measurements to be taken during the execution of an instrumented serverless function. A concept is presented to collect function metrics across function boundaries and reconstruct the temporal ordering of function calls using a distributed tracer. To view and analyze the results, an offline tool allows the user to perform user-defined analyses and visualize data.

We first give background information on serverless computing and discuss the execution environment, deployment, and trigger of a serverless function

using Amazon Web Services (AWS) Lambda and Google Platform Cloud (GCP) Functions. In Chapter 3, we introduce related work focusing on cloud provider solutions and work from the research sector. We then introduce the high-level design of the profiler. We detail the design, profiling format structure, and how the distributed tracer interconnects function calls. In Chapter 5, we offer more information about a possible implementation of the design in Python and address specifics in its deployment in AWS and GCP. We also show what metrics could be collected. In Chapter 6, we present measurements performed with the profiler with different functions and workflows and give details on the cost of the tool. We end the thesis with a conclusion and outlook.

Chapter 2

Background

In this chapter, we will introduce the reader to serverless computing. We present the different providers of serverless computing and discuss how they are structured, how functions are deployed, and how they can be triggered.

2.1 Serverless Computing

Serverless computing is a new emerging cloud computing and deployment model known as Function as a Service (FaaS) or Cloud Functions. Several leading cloud providers offer to run code in a serverless environment on their platforms. After the introduction of AWS Lambda in 2014 [2, 17], Google Cloud Functions [37], Azure Functions [48], and the open source platform Apache OpenWhisk [12] also followed with an implementation of the paradigm.

Unlike other cloud computing models, applications deployed in a serverless environment are divided into several individual functions. This allows in a traditional environment to allocate monolithic developed applications into fine-grained stateless functions [4]. These functions are triggered by different events, making serverless computing an implementation of an event-driven ideal. [54]. The functions are primarily developed separately, resulting in a high degree of flexibility. With the help of virtualization through containers, the individual functions are executed in an isolated environment. Therefore, the serverless concept allows high concurrency of functions, as multiple instances of the same functions can be activated as needed [63]. Scaling a function according to the function's load is usually handled automatically by the provider.

As the name Function-as-a-Service suggests, cloud providers focus on providing the function, abstracting the management of the runtime and underlying servers from the developer. This dramatically simplifies the provision of soft-

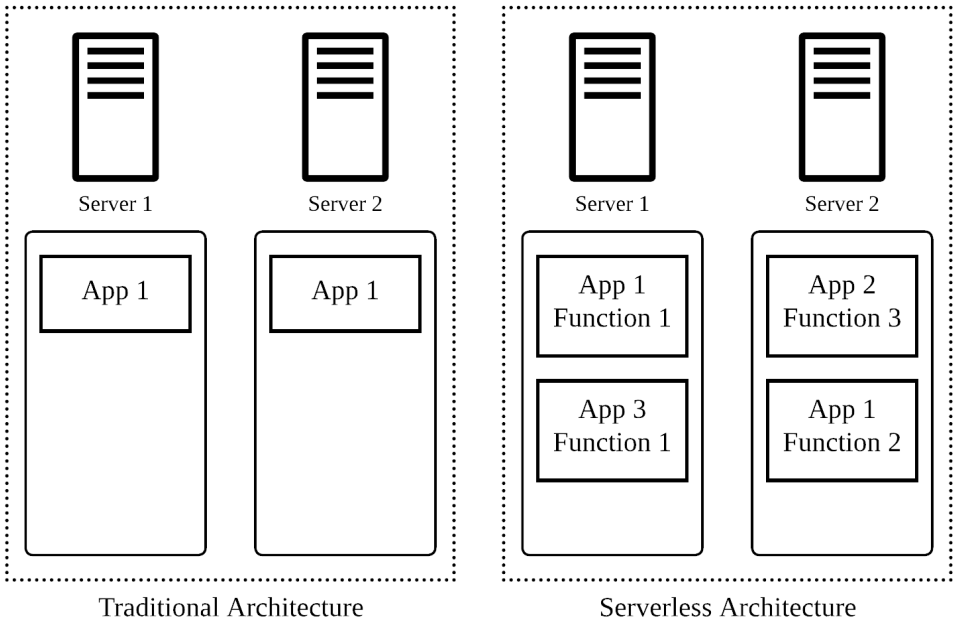


Figure 2.1: Comparison between traditional architecture and serverless. In the serverless model, several functions of different applications run on one physical machine. In the traditional architecture, the entire application is run in a container on different physical servers [62].

ware and leads to the high scalability of the functions. Users of the serverless model are also charged using the pay-as-you-go method, according to which only the resources used need to be paid for [4]. In other words, the user does not have to pay the components to make the server ”always-on” even if no requests are handled. For serverless to be profitable for providers, thousands of independent instances of functions are normally run on a physical server [63]. This guarantees high server utilization.

The developer is left with fewer configuration options compared to traditional models. Depending on the provider, the runtime, the timeout (i.e., the maximum execution time), memory limit, and the size of the ephemeral storage of a function can be selected. Billing is based on used execution time and memory consumption.

As noted, ”serverless” does not mean an application runs without a server. It refers much more to the fact that the management and provisioning of the servers are absent since this is completely taken care of by the provider.

In the following subsections, we will describe the serverless execution model in more detail by going into the execution environment, explaining how functions can be deployed, and finally, describing how functions can be triggered. We will focus primarily on AWS and GCP as cloud providers since the program presented in this thesis was tested with these.

2.1.1 Execution Environment

Runtime	AWS Lambda	Google Cloud Functions
NodeJS	12, 14, 16	6, 8, 10, 12, 14, 16
Python	3.6, 3.7, 3.8, 3.9	3.7, 3.8, 3.9
Ruby	2.7	2.6, 2.7, 3.0
Java	8, 11	11, 17
Go	1.x	1.11, 1.13, 1.16
.NET	Core 3.1, 6	Core 3.1
PHP	-	7.4, 8.1

Table 2.1: Available runtimes and versions in Google Cloud Functions [38] and Amazon Web Services Lambda [29]

The execution environment is the environment set up by the cloud provider into which the serverless function is invoked. Within this environment, the runtime environment requested for the function is initialized. Table 2.1 shows the different runtimes available in AWS and GCP. In addition to these predefined runtimes, AWS also supports user-defined runtimes. Each function is executed in an isolated and secure context [38, 18] and within a container.

Environment Life Cycle Each environment has a life cycle; the cycle phases depend on the provider. Across providers, it can be stated that the execution environment is not destroyed directly after the function has been performed. This ensures that the execution environment is not rebuilt entirely for subsequent calls of the same function.

A function call in an existing environment is also called a *warm-start*. In contrast, *cold-start* refers to the absence of the environment. A cold-start is always associated with higher latency because the FaaS platform must first spin up the environment before it can execute the actual function. In the case of AWS, the code must be downloaded first; then, the environment is loaded with configured runtime, memory, and parameters [19]. AWS calls this phase *Init* and divides it into three subphases: *Extension Init*, *Runtime Init*, and *Function Init*. The first set up all third-party extensions associated with the Lambda Function, the second load's resources are required by the runtime, and the latter executes the function's static code [18]. AWS limits the *Init* phase to 10 seconds.

For Cloud Function in GCP, cold-starts are described similarly; when executing a function without an existing environment, the code is first downloaded, and the requested runtime is loaded [38].

Neither AWS nor GCP indicates how long an environment is held before it is destroyed. However, Copik et al. showed in their paper that the function containers are kept active between 9 to 15 minutes, depending on the provider [5].

When a new function is deployed, the next call to that function is always a cold start [38, 18].

Listing 1 Local and global scope in Execution Environment

```
// Global scope
import numpy as np

def handler(*args, **kwargs)
    // Local scope
    pass
```

AWS and GCP distinguish between global and local scope in the function code. Global scope is all the code defined outside the function handler [38, 23]. This is loaded once per execution environment. Typically, imported libraries are found in the global scope, so they only have to be imported once. The local scope is the function handler reloaded on each execution. Listing 1 shows the difference using a Python function.

Summary of execution environment characteristics

Below, we will summarize the features of the Google Cloud Functions and AWS Lambda execution environments.

AWS Lambda AWS Lambda calls the function in a secure and isolated runtime environment. This manages all resources to execute the function. The user is offered the possibility to choose a predefined runtime (see Table 2.1) or to define it by a container. In the latter case, it is left to the user to decide which operating system to use. If a predefined runtime is chosen, Lambda uses its own Linux distribution, called "Amazon Linux," with preinstalled packages [1]. This supports the architectures `x84_64` and `arm64` [21].

A memory from 128 to 10240 MB can be allocated to a function.

The maximum configurable execution time is 900 seconds. The minimum is 3 seconds.

Lambda only allows writing access from the function to the folder `/tmp`. This can be configured from 512 up to 10240 MB. This is only reset in the next init phase.

[1] https://aws.amazon.com/amazon-linux-ami/2018-03-packages/

Lambda function generally has network access. Further steps are necessary if they are part of a VPC (Virtual Private Cloud) [27].

An file system can be attached to a Lambda function using the AWS service Elastic File System (EFS). A maximum of one file system can be connected.

An entry point can be defined that specifies where the function to be executed in the source code is located.

Google Could Functions Google Cloud Functions entirely manages the user's infrastructure, operating system, and runtime. Each function runs in its secure context, independent of other functions. Google supports the runtimes shown in Table 2.1; custom runtimes are not supported. The operating system is Ubuntu with pre-installed packages [2]; the version may differ from runtime to runtime.

A memory from 128MB to 8GB is allocated in discrete intervals.

The maximum execution time can be configured to 540 seconds. The minimum is 1 second.

Google Cloud Function allows writing access to all folders except the function code folder. The file system is an in-memory implementation, which means that stored data consumes the memory allocated for the function.

Every function has access to the public internet by default.

GCP is more restrictive than AWS in choosing an entry point. The platform forces the developer to have the method in a specific module depending on the runtime [47].

2.1.2 Deployment

To deploy an AWS Lambda function, there is an option to use a ZIP file or container image [25]. The first method uses a ZIP file consisting of the complete function code plus all dependencies. Lambda does not install any dependent package; all third-party packages must be included with the function code in the ZIP [24]. To deploy a container image, a container must be uploaded to ECR (Amazon Elastic Container Registry). An entry command is specified to invoke the Lambda function inside the container. All dependencies and the entire source code of the function must be present in the container.

To deploy a function in Google Cloud Platform, there is only the possibility to give the function's source code directly. For example, the source code can be provided by uploading a ZIP file or via a repository. The service generates

[2] https://cloud.google.com/functions/docs/reference/system-packages

an executable image for each new deployment, automatically uploaded to the container registry [40]. These images are generated with the help of Google Cloud Build.

2.1.3 Triggers

A serverless function can be triggered and executed in different ways. In most cases, the function is executed driven by an event. As execution forms, the synchronous and asynchronous kind is available. In a synchronous execution, the caller waits until the function has been executed, and the result of the called function is returned. In asynchronous execution, also known as fire-and-forget, the caller returns immediately and does not wait until the function has been called. The triggered function will eventually be executed in the future.

In AWS, many of the services offered by Amazon provide integration with Lambda, so many triggering events can be configured [32, 28]. Depending on the triggering service, the function is executed asynchronously or synchronously; AWS Lambda provides the ability to retry calls if they fail automatically [26]. A Lambda function is passed an event context following its triggering, which can vary greatly depending on the cause. Many services[3] pass a list of records to the function. The records can contain information about the triggering event, e.g., the object key of a changed object in S3 or the key of an entry in DynamoDB. Other call types, such as calling the function via the SDK or the AWS API Gateway, do not contain any information about the triggering event besides the general function payload. Amazon also provides the ability to route events from third-party platforms to a function via EventBridge [30].

Google Cloud Functions distinguishes between HTTP and events triggered by other Google Cloud Services [43]. A function can be performed via Cloud Pub/Sub, Cloud Storage, Firebase and Cloud Log. Cloud Pub/Sub is a service to distribute messages and data and also supports the possibility to call cloud function asynchronously [45, 13]. Event-driven calls receive an event context as a payload. This contains various metadata about the triggering event and information about the object in the case of cloud storage or the message in the case of cloud pub/sub [39].

AWS StepFunctions and GCP Workflows

AWS and GCP offer the ability to orchestrate various cloud products using workflows [34, 44]. AWS StepFunctions and GCP Workflows let users configure state machines, making it easy to construct complex business logic by connecting different serverless functions and other services. Both provide

[3] CloudFront, CodeCommit, DynamoDB, Kinesis, S3, SNS, SES, SQS [32]

simple control operations based on the state and result of previous executions, parallel executions, error control, and human interaction within the workflow. In the context of serverless functions, AWS StepFunctions and GCP Workflow can be seen as independent services that automate the triggering of functions based on states in the workflow.

Chapter 3

Related Work

This chapter will present existing solutions and works related to FaaS-Profiler shown in this thesis. We will first discuss solutions that cloud providers offer to their customers. Then, we will discuss different approaches to solutions from the research sector.

3.1 Cloud Provider Solutions

3.1.1 Amazon Webservices

Amazon Web Services offers several ways to display metrics and traces of a Lambda function. The essential services are AWS CloudWatch and AWS X-Ray: The former provides a logging service while the latter implements a distributed tracer.

AWS CloudWatch CloudWatch [16] offers integration to almost all AWS services and is also enabled by default for Lambda functions. CloudWatch aggregates log entries generated during an execution of a Lambda function. Each Lambda function is assigned a CloudWatch log group and a log stream for each active instance of the function. Output to the standard output is transferred to the logs. In addition to collecting logs, metrics are also generated. AWS Lambda automatically generates metrics about the function calls, e.g., number of calls, errors, and throttles, as well as performance-relevant units, e.g., duration, and information about the function's concurrency, e.g., number of concurrent executions.[1] CloudWatch collects data from Lambda at one-minute intervals per default.

[1] A complete list of metrics can be taken here: https://docs.aws.amazon.com/lambda/latest/dg/monitoring-metrics.html

AWS X-Ray X-Ray [20] is Amazon's implementation of a distributed tracer to analyze and debug distributed systems. It provides the ability to visualize a service map to trace the end-to-end journey of a request. AWS Lambda includes integration for X-Ray, but this must be enabled manually and may require changes to the source code, for example, to instrument the method. In X-Ray, a trace is referred to as a request with multiple services; a trace has numerous segments, which are layered into various subsegments. In the case of a Lambda call, a segment is created that is divided into three subsegments *Initialization*, *Invocation*, and *Overhead*. *Initialization* refers to the section to load the Execution Environment, *Invocation* is the actual calling of the function, and *Overhead* is the period between the time the Runtime Environment sends the response and the signal for the next call [31].

AWS X-Ray for Lambda uses a daemon to manage trace records. Records are first sent to the daemon via UDP traffic, cached, and then sent to the X-Ray as a batch to reduce latency. The daemon is fully managed, and the user cannot influence the behavior.

Based on the collected records, X-Ray provides the possibility to visualize a service map of requests and analyze bottlenecks and errors.

3.1.2 Google Cloud Platform

Under the Cloud Operations service, Google Cloud offers various ways of gaining insights into ongoing cloud processes. Operations are the central location in Google Cloud, where log entries are collected and performance metrics are processed. These include logging, monitoring, tracing, debugging, profiling, and error reporting [41].

Cloud Monitor collects metrics, metadata, and event details about applications and systems in the cloud, including cloud functions. The data is automatically processed and visualized in the form of dashboards. For example, each cloud function's calls per second, execution time, memory usage, and active instances are displayed.

Log entries of all services are collected via cloud logging. All entries output via standard output for Google Cloud Functions is displayed as a log entry. The entries are grouped by function, execution, and trace.

With Cloud Trace, Google realizes an implementation of a distributed tracer. The tool collects latency information of an application from user request to completion and visualizes the different components as a trace. The data is automatically collected and processed in near real-time.

3.1.3 Comparison with FaaS-Profiler

As seen, Google Cloud Platform and Amazon Web Services each offer similar services to gain insight into the serverless function. One limitation of their services is that they offer black-box metrics primarily and provide little information about the performance of a particular source code. In addition, the solution is provider-specific, which makes it challenging to compare performance across platforms. Last but not least, as pointed out in the paper by Wei-Tsung Lin et el., AWS X-Ray works with sampling and UDP packets to capture trace data, which may not guarantee complete insight into the trace [52].

FaaS-Profiler works platform independently, allows easy addition of measurement data on application level, and collects the same profiling format for all platforms.

3.2 Research Systems

Let's Trace It: Fine-Grained Serverless Benchmarking using Synchronous and Asynchronous Orchestrated Applications Joel Scheuner, et al., present in their paper [57] ServiBench, an application-level benchmark suite to perform end-to-end experiments with serverless functions and realistic invocation patterns. ServiBench operates as an offline tool especially tailored for AWS Lambda, which downloads the collected log files for called Lambda functions and reconstructs a temporal causal ordering between function calls. ServiBench handles both asynchronous and synchronous calls. Based on the X-Ray log data, a graph representing the causal relationship between calls is used to calculate a critical path so that the end-to-end latency of a request can be broken down. The happened-before order and the definition of an asynchronous call are made by temporal arguments alone; a tolerance was introduced to respect clock differences between calls. The presented tool breaks down the latencies of an application with real-world call patterns with synchronous and asynchronous calls. However, the tool is heavily dependent on AWS X-Ray, and the construction traces are based only on temporal reasoning.

Tracking Causal Order in AWS Lambda Applications Wei-Tsung Lin et al. present GammaRay [52] in their paper, a tool to track the causal ordering of Python applications in AWS Lambda. GammaRay was developed with the background of tracking and connecting asynchronous and synchronous calls to Lambda functions. The tool was split into a library to provide runtime support for Lambda functions and an offline tool that post-processes recorded events through the library. To trace outgoing requests to a lambda function or to collect information that can help to establish causal order later, the paper presents three ways: First, dynamically patching the AWS SDK, then

statically patching the AWS SDK, and reading and analyzing AWS X-Ray and CloudWatch entries. GammaRay uses a transactional database to record events and rework them offline in append order. For all three variants, the overhead was evaluated with mirco-benchmarks. The static instrumentation of the SDK was shown to produce the most memory overhead, followed by dynamic patching and the X-Ray variant.

Tracing Function Dependencies across Clouds Wei-Tsung Lin, et el., present a tool called Lowgo [51] to record causal relationships between functions in serverless applications. Lowgo was developed with the background of also establishing an ordering of applications operating in different clouds, thus providing a cross-cloud tracing tool. Lowgo automatically intercepts requests from the cloud SDK to be informed about possible invocation origins. These events are processed in a Lowgo instance located in each cloud. The tool does not need to record events in a database, which reduces overhead compared to GammaRay [52]. The instances cache the entries until the position in the trace can be found for the record. Lowgo replicates the records intercepted in different clouds across instances to create a consistent, distributed log. The authors could show in benchmarks that Lowgo can show the causal relationships, and the overhead in execution time is between 2-12%.

Chapter 4

Design

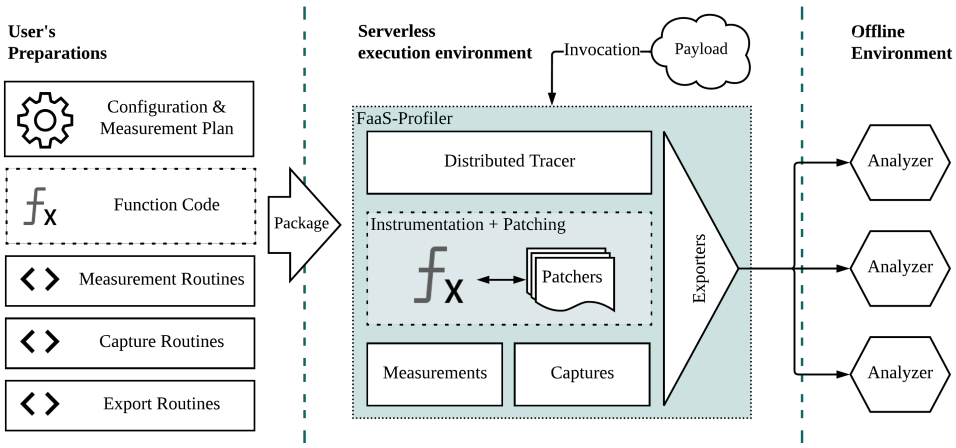


Figure 4.1: FaaS-Profiler Design Overview. It shows the different phases of using the framework. First, a configuration and measurement plan is defined, packaged, and deployed together with the instrumented function code and measurement/capture routines. In the serverless execution environment, the function is executed with active patchers. The distributed tracer handles incoming and outgoing requests while the measurement and capture routines collect data. Finally, these are exported so the analyzers can process the collected data offline.

In this chapter, we will describe the high-level design of the FaaS-Profiler profiling tool in more detail. The tool was developed with the ulterior motive of being independent of the chosen programming language and cloud provider. Therefore, the general structure and design idea will be discussed in this chapter. In contrast, in Chapter 5, implementation details in Python on the cloud platforms Amazon Web Services and Google Cloud Platform will be addressed.

In short, FaaS-Profiler provides an open and easily configurable framework that allows users to collect data and metrics about any serverless function

and then analyze and visualize them. In addition to collecting this data, the profiler has a distributed tracer that also allows metrics to be obtained across interacting functions.

The tool comprises two main components: A client library, hereafter referred to as *Instrumenter*, and a visualization and post-processing tool, hereafter referred to as *Visualizer*.

As shown in Figure 4.1, three phases are required to apply the FaaS-Profiler to any serverless function successfully. In the beginning, there is the phase of preparation for the user. Defining what data and metrics should be collected and where they should be exported is necessary. In addition, the *Instrumenter* must be packaged and deployed with the instrumented function code. In phase two, the function is executed in the serverless execution environment. The *Instrumenter* can now process incoming and outgoing requests in the tracer and perform routines to collect data. This data is exported through an exporter. In the final phase, the *Visualizer* reads this exported data, performs a postprocessing routine, and visualizes the data in the form of a dashboard. The calculations of the *Visualizer* are not performed in the cloud and can be executed locally and offline on the user's computer.

The chapter is structured as follows: We will first define the profiling format (Section 4.1) that the *Instrumenter* generates as output and the *Visualizer* analyzes as input. Then, in Section 4.2, we will discuss the *Instrumenter's* structure in more detail, explaining which part of the program collects which data.

4.1 Profiling Format

This section defines the profiling format automatically generated by the *Instrumenter* for each instrumented serverless function. If configured, this is then exported to the *Visualizer* for further analysis. The format was developed with the following four main goals:

- **Uniqueness**: Unique record is exported for each execution of a function.
- **Generalization**: The profiling format defines characteristics for a function independent of provider and runtime.
- **Modularity**: Any additional data to be exported can be defined.
- **Traceability**: Information that should later help to arrange the function call in a trace of other serverless function calls is stored.

A concrete instance of the profiling format contains, among other things, information to later sort the arrangement of invocations and construct a trace. We, therefore, call this instance a *Trace Record* and define it as follows:

Definition 4.1 (Trace Record) A *Trace Record* characterizes a single execution of an instrumented serverless function. A record has a unique ID and is composed of different contexts:

- *Function Context* (Section 4.1.1) gives general information about the function itself and the function call.
- *Tracing Context* (Section 4.1.2) gives IDs to the current trace.
- *Inbound Context* (Section 4.1.3) characterizes the request that led to the function's execution.
- *Outbound Contexts* (Section 4.1.3), possibly several, characterize all requests initiated by the function itself.
- *Record Data* (Section 4.1.4) is a list of various collected function metrics.

The data structure in Listing 2 represents a *Trace Record*.

Listing 2 Trace Record Struct Definition

```
struct TraceRecord
    id: UUID
    function_context: FunctionContext
    tracing_context: TracingContext
    inbound_context: InboundContext
    outbound_contexts: List[OutboundContext]
    data: List[RecordData]
```

As seen from Definition 4.1, the *Trace Record* consists of different contexts, each of which serves other purposes. In the following, we will go into more detail about the components of the *Trace Record*.

4.1.1 Function Context

The *Function Context* collects information about the executed function and the execution itself.

Definition 4.2 (Function Context) The *Function Context* exports information about the entire execution of the function and about the environment in which it was executed. Listing 3 defines the data structure that a *Function Context* follows. FaaS-Profiler creates this context about each call, guaranteeing its existence.

Listing 3 Function Context Struct Definition

```
class FunctionContext
    provider: Provider
    region: Region
    runtime: Runtime

    function_name: str
    handler_name: str

    invoked_at: datetime
    handler_executed_at: datetime
    handler_finished_at: datetime
    finished_at: datetime

    max_memory: int
    max_execution_time: int

    has_error: bool
    error_type: str
    error_message: str
    traceback: List[str]

    arguments: dict
    environment_variables: dict
    response: Any
```

Remark We want to discuss the difference between function names and handler names. The function name is the user's name for a particular serverless function. This name is entirely user-definable (except for provider-dependent restrictions) and is generally independent of the source code defined for the function. The handler name, on the other hand, is the name of the function in the source code to be executed for the function, so it is dependent on the provided source code and can be arbitrarily changed for a function if the code changes.

Function Key The Function Context contains details about the function in general, which remain the same for each function call. These are the *Provider*, *Runtime*, *Region*, and *Function Name*. Since the cloud providers ensure that no function in the same region has the same name, a *Function Key* can be defined:

Definition 4.4 (Function Key) The Function Context automatically provides a property named *Function Key*. This is composed of the *Provider*, *Region*, and

Function Name. The function key uniquely describes a serverless function across multiple providers and regions. Its existence is guaranteed.

The function key can be used to compare or group two different trace records of the same serverless function.

Timing Properties In addition to the general function properties, the function context characterizes the current execution. `invoked_at`, `handler_executed_at`, `handler_finished_at` and `finished_at` provide information about the duration of the function. The section between `handler_executed_at` and `handler_finished_at` indicates the time the instrumented function took. `invoked_at` and `finished_at` denotes the section that the entire execution took, i.e., instrumented function plus the overhead of the profiling tool.

Configuration The properties `max_memory` and `max_execution_time` gives information about the current configuration in which the function is executed.

Error Evidence The Function Context collects simple information about failed executions. `has_error` indicates whether the execution of the instrumented function ended with any error. If so, `error_type` and `error_message` contain the error type and message, respectively. Additionally, `traceback` exports the last operations of the function.

Payload and Response With the properties `arguments` and `environment_variables`, we summarize all the data passed to the function at the beginning of the execution. `response` is a wildcard field that stores all possible return values of the function.

Remark As presented, the *Function Context* contains many values, which probably inflates the size of the trace record. The modularity of the *Instrumenter* allows disabling all fields in the configuration except those defining the *Function Key* and the *Timing Properties*. This reduces the overhead in the runtime. By default, all fields are disabled except for the exceptions mentioned.

4.1.2 Tracing Context

Next, we define the *Tracing Context* used by the Distributed Tracer (see Section 4.2.6) to assign records to a trace.

Definition 4.6 (Tracing Context) The tracing context consists of up to three IDs and uniquely describes a record in a trace. It is created automatically with each function call and its existence is guaranteed. The data structure in Listing 4 defines a tracing context.

Listing 4 Tracing Context Struct Definition

```
struct TracingContext
    trace_id: UUID
    record_id: UUID
    parent_id: Optional[UUID]
```

Note that the Trace ID and Record ID always exist, while a Parent ID does not necessarily have to exist.

4.1.3 Inbound & Outbound Contexts

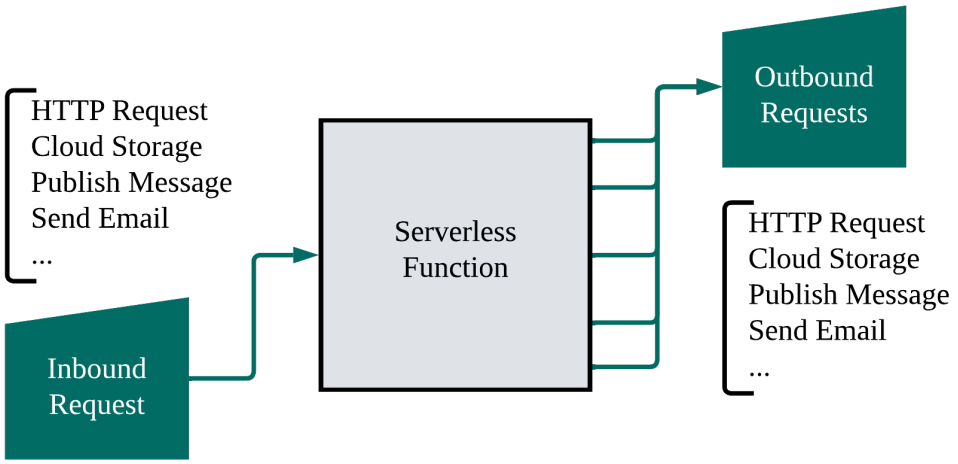


Figure 4.2: Difference between inbound and outbound requests. The inbound request leads to the execution of the function, while the process initiates outbound requests.

This section deals with the characterization of requests in the serverless environment. Figure 4.2 schematically shows the difference between an incoming and outgoing request in the context of a serverless function. The profiling format of the FaaS-Profiler generalizes this as a *Request Context*, distinguishing between an *Inbound* and an *Outbound Request Context*.

An *Inbound Context* characterizes the request that led to the triggering of the instrumented function. For example, this can be a simple HTTP trigger or a file upload to a cloud storage. With the inbound context, we thus generalize the trigger that triggered this function.

An *Outbound Context* is created when the current function triggers an operation of a service during its execution. For example, when the function uploads a file to storage, an outbound request is made to the cloud storage. Information about this operation is stored in the outbound context.

Both requests are generalized in the profiling format as *Request Contexts*. We define it as follows:

Definition 4.7 (Request Context) A *Request Context* contains information about a made request. With its help, the origin and reason of the request can be traced back, and information on the chronology can be provided. In addition, values have been attached that help to identify a particular request uniquely. The presented data structure in Listing 5 defines a *Request Context*.

Listing 5 Request Context Struct Definition

```
struct RequestContext
    provider: Provider
    service: Service
    operation: Operation
    trigger_synchronicity: TriggerSynchronicity

    identifier: dict
    tags: dict
```

Request origin and reason `Provider`, `Service`, and `Operation` characterize the origin and reason of the request. The three values are dependent on each other and limit the possible values. The choice of provider restricts the choice of service since, for example, only services from AWS are now available for selection. Service, in turn, determines the operation. Assuming that the service is AWS S3, for example, only ”Object created” or ”Object deleted” remain available for selection as operations.

`TriggerSynchronicity` indicates whether the request was asynchronous and synchronous.

Request metadata `Identifier` and `tags` are key-value mappings that can store arbitrary information about the request. Especially `identifier` should be emphasized because these values should help to describe a request uniquely.

Example 4.8 (Identifier for AWS S3) To explain inbound and outbound context more efficiently, we give an example of a workflow in AWS Lambda. Suppose we have deployed two Lambda functions. Function *Upload* uploads an image to a bucket, and function *Process* is executed by Lambda when a new image is uploaded to that bucket.

Function *Upload* makes an outbound request, namely `S3::UploadFile`. FaaS-Profiler detects the outbound request and saves the following request context for the function:

```
provider: AWS
service: S3
operation: UploadFile
trigger_synchronicity: SYNC

identifier:
    bucket_name: MyBucketName
    object_key: example_file.txt
    request_id: 175f2e7a-d3ad-4b1e-9b2a-548142ac5eb2
```

Function *Process* detects an inbound request, namely the trigger of `S3::UploadFile`. FaaS-Profiler extracts information about the inbound request and stores the following inbound request:

```
provider: AWS
service: S3
operation: ObjectCreated
trigger_synchronicity: ASYNC

identifier:
    bucket_name: MyBucketName
    object_key: example_file.txt
    request_id: 175f2e7a-d3ad-4b1e-9b2a-548142ac5eb2
```

How FaaS-Profiler precisely extracts the characteristics of the request is explained in Chapter 5 about implementation. We will see later, in Section 4.2.6, how the distributed tracer uses this information to connect two independent function calls.

Since a function execution has been performed only by a specific trigger, there is, at most, one inbound context. Since a function can call any number of other services, there can be any number of outbound contexts for a record.

4.1.4 Record Data

We call *Record Data* all the results collected during the execution of the instrumented function.

Definition 4.9 (Record Data) Record data is a simple list of function metrics collected during the function's execution. Each entry in the list has a unique name associated with it, which facilitates later access to the data.

Example 4.10 (Trace Record for AWS) Assume the following Python function was deployed to AWS Lambda in region `eu-central-1` with the name `my_lambda_function` and then executed by a Lambda trigger. The functions uploads a JSON object to S3.

```
import faas_profiler_python as fp

import json
import boto3

@fp.profile()
def upload_handler(event, context):
    s3 = boto3.client("s3")
    file_body = json.dumps(
        {"message": "Hello Lambda"}).encode('utf-8')
    s3.put_object(
        Bucket="my-bucket",
        Key="my-object.json",
        Body=file_body)
    return {
        "message": "Uploaded"
    }
```

FaaS-Profiler automatically generates and exports the following trace record, including detected inbound context for Lambda and outbound context for S3.

```
tracing_context:
  record_id: 17b693c9-572f-4829-b63b-f3fd43968c55
  trace_id: d891a835-3fe7-4bc7-ac3c-fd68c03bb106
function_context:
  provider: aws
  handler: handler
  function_name: my_lambda_function
  region: eu-central-1
  runtime: python
  invoked_at: '2022-09-30T10:39:12.421205'
  handler_executed_at: '2022-09-30T10:39:12.422787'
  handler_finished_at: '2022-09-30T10:39:12.920682'
  finished_at: '2022-09-30T10:39:12.924835'
  max_memory: 1024
  max_execution_time: 5
  environment_variables: {...}
```

```
  arguments: {...}
  has_error: false
inbound_context:
  provider: aws
  service: lambda
  operation: invoke
  trigger_synchronicity: SYNC
  invoked_at: '2022-09-30T10:39:12.422098'
  identifier:
    function_name: my_lambda_function
    request_id: 6c502e72-749f-4187-ba4c-75ea09d4f2af
outbound_contexts:
  - provider: aws
    service: s3
    operation: ObjectCreated
    trigger_synchronicity: SYNC
    invoked_at: '2022-09-30T10:39:12.432098'
    finished_at: '2022-09-30T10:39:12.482098'
    identifier:
      bucket_name: my-bucket
      object_key: my-object.json
      request_id: 39f6f4bb-9c65-422b-b438-68a54654887b
```

4.2 Instrumenter

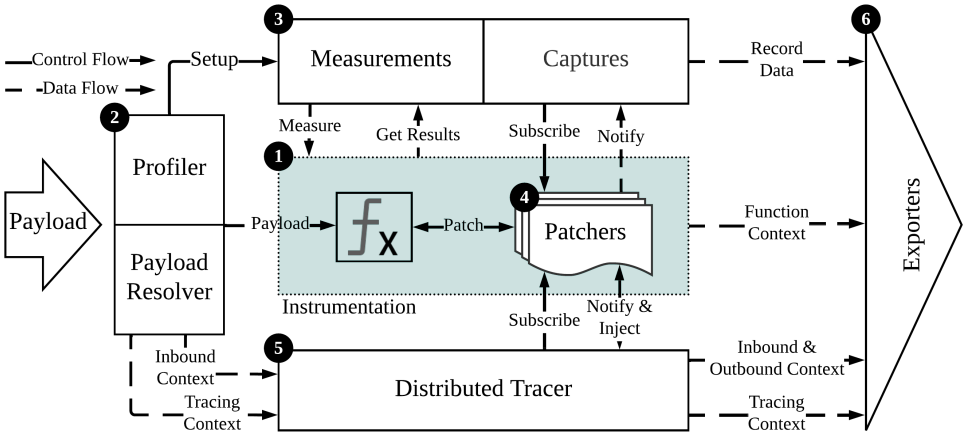


Figure 4.3: Schematic structure of the FaaS-Profiler Client. The components shown run in the execution environment of the serverless function to be profiled.

The *Instrumenter* is responsible for instrumenting, collecting functional metrics, and exporting these metrics. It provides a configurable framework to facilitate the addition of custom components.

This section focuses on the design structure of the instrument, which is shown in Figure 4.3. After discussing in the previous Section 4.1 what our format looks like, which will be used for profiling and tracing a serverless function, we now introduce the *Instrumenter*, which is executed in the same environment as the instrumented function and generates this format.

We introduce the payload, Section 4.2.2, of any serverless function and present how it can be represented uniformly and what information can be extracted from it. We will show in Section 4.2.3 how routines are defined to collect function metrics. We also introduce a patching framework, Section 4.2.5, that allows modification of arbitrary functions at runtime to collect metrics about their execution and intercept and record outgoing requests. In the last Section 4.2.6 we introduce our version of a distributed tracer. A distributed tracer is a tracer that tries to establish temporal order in a distributed system, such as serverless workflows. We will see how FaaS-Profiler uses the Tracing Context (section 4.1.2) and the Request Contexts (Section 4.1.3) to reconstruct an order.

4.2.1 Instrumentation

To activate the profiler and instrument, ❶ in Figure 4.3, a function, the function to be examined is passed along with arguments to the profiler entry point. The client can load and set up all necessary resources before executing the function. The instrumented function is then executed by the profiler, while all possible exceptions are caught and documented during execution. Also recorded is how long the function took to complete. This information is stored in the *Function Context*, Section 4.1.1.

4.2.2 Payload

The following section covers ❷ of the overview shown in Figure 4.3. Recall that the client's goal is to generalize all possible providers and programming languages. Serverless functions can be called in different ways and with different arguments. Therefore, we define the term *Payload*.

Definition 4.11 (Serverless Function Payload) The payload of a function is defined by all the data available to it before it is executed. More precisely, these are the functions' arguments, environment variables, headers, and all data that can be retrieved without using third-party software.

Note that this term is intentionally comprehensive. This is necessary because the payload can differ significantly from provider to provider and from trigger type to trigger type.

A payload representation must be defined to understand and break down the structure of a particular cloud platform. Each payload representation follows the interface in Listing 6.

Listing 6 Payload interface

```
interface Payload
    public procedure extract_tracing_context
    public procedure extract_inbound_context
end
```

As can be read from Listing 6, a concrete payload representation extracts an inbound context, which should give information for the request that leads to the execution of the function, and a tracing context, which maps the function call to a trace. The both extracted contexts are passed to the distributed tracer (Section 4.2.6).

Correctness of the extracted tracing context

Extracting the tracing context has its limitations. In general, there is no guarantee that a tracing context exists in the incoming payload, and if it exists, it is correct. An example is asynchronous triggers: As we will see later in the discussion of the distributed tracer, Section 4.2.6, for specific asynchronous triggers, it is impossible to send the tracing context to the triggered function. The triggered function does not know at the time of execution that it belongs to the trace of the triggering function. As a result of the absence of the tracing context in the payload, the function creates a new context and treats itself as the root of a new trace. The correct context, in the sense of connecting the two calls, can only be clarified in the post-processing of the *Visualizer*; hence the note that its context may be incorrect at the time of execution.

4.2.3 Function Metrics

Generally speaking, *Function Metrics* are the data that should give the user of FaaS-Profiler more insight into the execution of the serverless function. FaaS-Profiler roughly distinguishes between two different types of function metrics: *Measurements* and *Captures*.

Measurements allow the recording of a metric. This can, for example, be dependent on the time of the function execution and record the memory consumption of the function by measuring points. Or measurements can be taken directly before and after the function to calculate a difference introduced by the function.

Captures, on the contrary, allow notifying when the instrumented function has invoked a specific function. For example, if we want to know how long and often a cloud storage upload was made, we tell the FaaS-Profiler to capture the upload method. Based on the notification, we can calculate cloud-specific metrics.

In the overview Figure 4.3 ❸ shows this process of collecting metrics.

Measurements

Measurements are program routines designed to facilitate the collection of data of interest during function execution. The client allows you to perform two different measurements: *Periodic Measurements* and *Simple Measurements*, where each periodic measurement is a simple measurement but not vice versa.

Definition 4.12 (Simple Measurement) A simple measurement is a routine triggered once before the instrumented function is executed and after the function is executed. Simple measurements are performed in the same process as the function.

Simple measurements follow the interface in Listing 7.

Listing 7 Simple Measurement interface

```
interface Measurement
    public procedure initialize
    public procedure start
    public procedure stop
    public procedure deinitialize
    public procedure results
end
```

`initialize` can be used to introduce data structures and to prepare systems for measurement, e.g., importing load-expensive third-party libraries. This function is also passed user defined parameters for customization. `deinitialize` can be used again to free the resources.

`start` and `stop` are commands to start and stop the measurement respectively. They are called immediately after function start or function stop.

The *Instrumenter* makes the following guarantees to the measurement classes:

- `initialize` is the first method that is executed. There is no guarantee of the success of this execution.
- `start` is called after all other measurements have been initialized.
- `stop` is called after all other measurements have been started.

- `deinitialize` is the last method to be called. Before that, all measurements were stopped.

For example, snapshot measurements can be performed with simple measurements. For this purpose, a measure of a metric of interest is performed before the start of the function and a measure after the completion of the function. A difference can be calculated with these measurement points, which was caused by the function. In Chapter 5, about implementing the FaaS-Profiler, we will see how we used this concept, for example, to measure I/O counters.

To collect time-continuous data, there is a possibility to perform periodic measurements. As the name suggests, these measurements are triggered periodically during the execution of the function instead of only at the beginning and end of the function, as with simple measurements.

Definition 4.13 (Periodic Measurement) Periodic measurements are extensions of Simple measurements. These are triggered at equal intervals during the execution of the instrumented function. The default delay between two measurement calls is 0.1 seconds. The user can change this delay. Periodic measurements are performed in parallel in a child process.

Periodic measurements follow the interface in Listing 8.

Listing 8 Periodic Measurement interface

```
interface PeriodicMeasurement extends Measurement
    public procedure measure
end
```

The motivation for introducing the periodic measurements is the following: For capturing some metrics, it is too coarse-grained to take only one measurement point at the beginning and the end of the function. An example of this is the memory consumption of longer-running functions. To record the behavior of the allocated memory space during the execution, it is necessary to be able to add further measuring points parallel to the running of the function. If we were to take measurement points only at the beginning and end instead, we would get an incomplete picture of memory consumption, since memory could be freed up again during execution. Periodic measurements fulfill precisely this purpose.

Besides the guarantees of the simple measurements, the following guarantees are made:

- `measure` is always called after `start` and before `stop`. No guarantee is made about the number of calls.

Process synchronization of periodic measurements As explained in Definition 4.13, all periodic measurements are executed in another process, hereafter

referred to as *Measuring Process*, to perform measurements in parallel with the main process in which the function is executed. FaaS-Profiler makes the following guarantees by process synchronization of the two processes:

- `initialize` and `start` a periodic measurement, are executed and finished before the instrumented function starts. In other words, the primary process waits for the measuring process until all periodic measurements are started.
- `measure` may be executed again after the function is finished due to polling delay.
- `stop` and `deinitialize` are called after the main process has reported that the function has finished executing.

Captures

Captures are routines designed to facilitate documenting function calls as the instrumented function is executed. Capture routines can be used to capture cloud-specific performance metrics. One application area, for example, would be to record how long and how often the function makes API requests to other services to identify possible bottlenecks. Capture routines are, therefore, always suitable if you want to observe the behavior of specific function calls.

Definition 4.14 (Capture) Capture is a routine that is notified each time a function of interest is called during the execution of the instrumented serverless function.

Captures follow the interface in Listing 9.

Listing 9 Interface for capture routines.

```
interface Capture
    public procedure initialize
    public procedure start
    public procedure capture
    public procedure stop
    public procedure deinitialize
end
```

`initialize`, `start`, `stop` and `deinitialize` have the same function as in simple measurements. `capture` is then called automatically if the method of interest is called.

In general, there are several ways to intercept calls to specific functions. Mainly the choice of these possibilities is strongly limited by the selection of a programming language in which the client should be available. Here

we present a solution using *Monkey-Patching*, described in more detail in Section 4.2.6.

Order of measurement routines

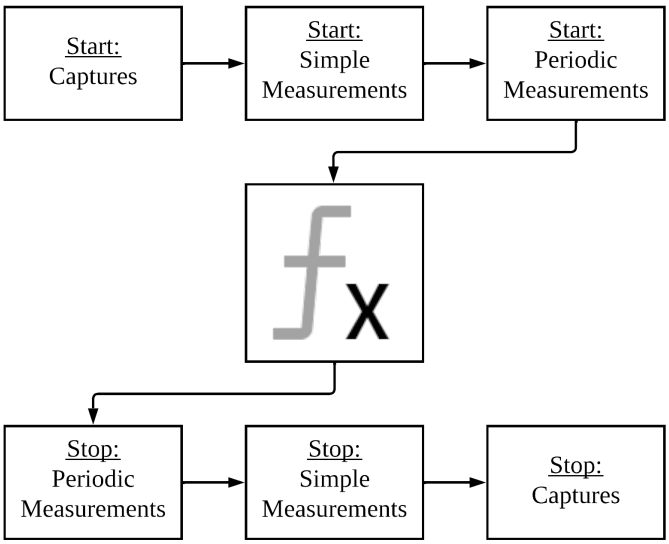


Figure 4.4: Order in which captures, simple and periodic measurements are started and stopped.

Figure 4.4 shows how measurement and capture routines are started and stopped in the FaaS-Profiler. The order was chosen so that the measurements influence each other the least. For example, since periodic measurements can be used to query the memory usage over the time of execution, we want to start them last so that the setup of the other routines is not included in the data. For the same reason, we start and stop captures first and last, respectively, since preparing to intercept function calls produces more overhead.

4.2.4 Exporters

The task of an *Exporter* ❻ is to export the collected data and thus store it for later analysis. Each defined *Exporter* is given the *Trace Record* created by the execution of the instrumented function. For example, the *Trace Record* can be written as JSON in a cloud storage bucket or stored in a database. The final medium used for export is configurable and modular interchangeable. The *Instrumenter* guarantees that the export is the last action performed before returning the result of the actual method.

Exporters follow the simple interface in Listing 10.

Listing 10 Exporters interface

```
interface Exporter
    public procedure initialize
    public procedure export
end
```

FaaS-Profiler currently defines a *Record Storage* interface that can be used optionally for ease of use. The interface serves as a kind of contract between the *Instrumenter* and the *Visualizer* and guarantees that the *Visualizer* will find records stored by the *Instrumenter*.

4.2.5 Patching Framework

The FaaS-Profiler *Instrumenter* provides a framework, shown as ❹ in Figure 4.3, to quickly patch any functions. We will first describe the general notion of *Patching* (also called *Monkey-Patching*). Then we will explain the framework, which plays a vital role in capturing and tracing functions.

Monkey-Patching

Monkey patching is a technique that can be applied to dynamic programming languages such as Python and Ruby. The idea that it is possible to add behavior to an existing object at runtime to meet some requirement that originally the type did not meet [14].

Assume that the method `foo` was defined in the module `bar` in the original source code. To patch the method `foo` now dynamically with another method, `monkey_foo`, the address of the method definition `foo` is replaced by the method definition `monkey_foo`. If the method `foo` is now called, not the source code of `foo` is executed, but that of `monkey_foo`. The method was thus replaced in runtime by another without changing the original source code.

Patching is mainly used in unit tests (i.e., in Python's mock library [11]) and editing and extending third-party software.

The Framework

In general, any code of the FaaS-Profiler must never change the behavior of the instrumented function. In other words, the serverless function must behave exactly identically with and without the FaaS-Profiler. For this reason, we never completely replace a function of interest with another function; instead, the function is replaced with a *Wrapper*. A wrapper executes logic before and after the patched function and returns the return value of the original function. So for the original method, the patch is invisible.

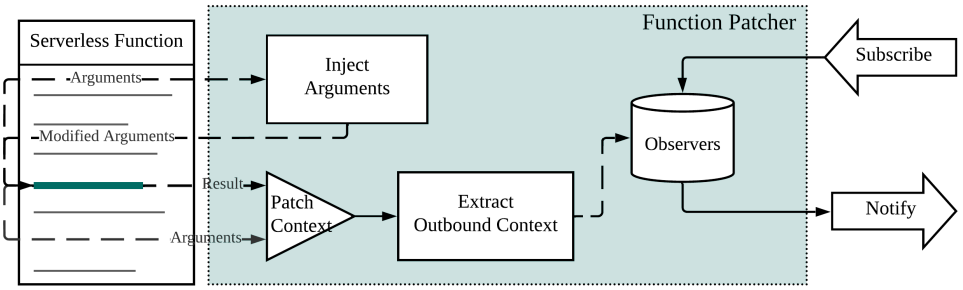


Figure 4.5: Function patcher in FaaS-Profiler with event observer pattern. A function patcher is defined for a specific function. FaaS-Profiler automatically exchanges the function with a wrapper to catch arguments to the function and return values. After determining an outbound context for the particular call, all interested observers are notified. If needed, the function arguments are injected before the original function is called.

To patch an arbitrary function (for example, an API call of a third-party library), a *Function Patcher* is defined for the library function. A function patcher is a class with the interface in Listing 11.

Listing 11 Function Patcher interface

```
interface FunctionPatcher
    public static property module_name
    public static property function_name

    public procedure activate
    public procedure deactivate

    public procedure register_observer
    public procedure set_data_to_inject

    public procedure extract_context
    public procedure inject_payload
end
```

The properties `module_name` and `function_name` tell the framework in which module which function should be patched. After checking whether a function patcher has already been defined for the same function, the function is replaced by a wrapper at runtime, as described above.

The Patcher of a function performs two main tasks, which must be defined individually by the user for each patched function: Firstly, the extraction of information about the function call and, secondly, the modification of function parameters. The patching framework has been designed to simplify the definition of these methods. The modality allows multiple patcher

routines to be applied to different target methods. Moreover, the framework takes care of extracting the arguments and results of the target method and passes them uniformly to the concrete patcher.

Information Extraction An individual patcher's `extract_context` method defines the logic to extract information for a function call. Given the original method and arguments, as well as the result after execution of the patched function, a context can be created that provides further information about the execution. This can be, for example, the request ID of an API to a service or the filename of a file upload.

Parameter Modification Sometimes, it is necessary to modify the function arguments dynamically before executing the patched method. We will see later, in Section 4.2.6, that the distributed tracer makes elementary use of this possibility. The method `inject_payload` of an individual patcher defines how to modify the function's arguments. So if data was set to inject with the method `set_data_to_inject`, the method `inject_payload` tries to add it to the original arguments if the patched function allows it. Note that `inject_payload` is always called strictly before calling the original function.

With the method `register_observer`, other parts of the tool (e.g., captures and the distributed tracer) can register themselves as interested in the patched function according to the *Event-Observer pattern*. The framework automatically calls the observer with the extracted information; it then remains for the observer to take further steps with the data. The critical point of constructing the function patchers in this pattern is that we want to ensure only one active patcher for each function. If more than one party is interested in that patcher, they can register as observers. The reason for this is trivial since we can only replace one function with one another function. Furthermore, functions are only patched if the patcher holds a lock for the function to avoid possible problems with concurrent programs.

4.2.6 Distributed Tracer

As explained in the Background section, Chapter 2, serverless application can be composed of multiple serverless functions.

FaaS-Profiler is designed to collect functional metrics across multiple serverless functions. To achieve this, our tool implements a distributed tracer, shown as ❺ in Figure 4.3. A distributed tracer traces execution threads in a distributed environment. Here, distributed refers to the many independent services that communicate and interact with each other through events. In terms of the serverless execution model, we want to be able to trace which serverless function led to the execution of another serverless function.

To fulfill this goal, FaaS-Profiler uses the extracted information in the *Tracing* and *Request Contexts*, Section 4.1.3, (inbound and outbound contexts) to reconstruct a partial temporal ordering between function calls. The distributed tracer works actively in the *Instrumenter* on the one hand; on the other a hand a postprocessing routine in the *Visualizer* is needed to record all traces.

We start with a problem definition and first approaches how to order function calls. Followed by a theory part where we explain when two function calls are comparable in a partial order. Last but not least, we present insights into our concrete version of the tracer.

Problem statement and first approaches

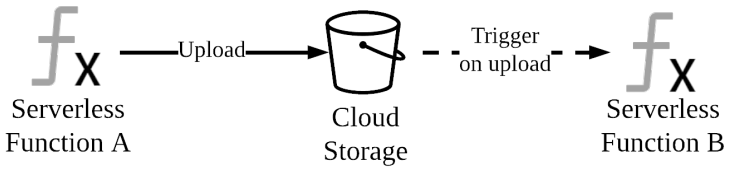


Figure 4.6: Depicted is a serverless workflow where function A uploads a file to cloud storage, and function B is triggered for each new upload.

The basic idea of a distributed tracer is the following: We want to find a way to tell a serverless function the reason and origin for its execution. Assuming that it would be possible for each function to know precisely what other execution of a serverless function led to the current execution, one can accurately trace the functions. Here, the function with no predecessor is the root of the trace.

A simple and naive solution would be that each function that calls another serverless function sends along data that makes it clear that the called function is part of a trace. This is precisely what the tracing context, Section 4.1.2, can be used for. The calling function sends its Record ID R and Trace ID T to the called function. If the called function now sees these IDs, the function knows that it belongs to the trace with ID T and that its parent has the Record ID R. It is now possible to connect the two calls in this way. This relatively simple possibility solves most cases, but not all, as we will explain now.

Recall that many different events can execute serverless functions. It is important to note that functions may not be directly triggered by other functions but by services that act as a "middleman". By this we mean, for example, an execution resulting from an upload to S3, with S3 as the middleman, or a publication of a new message in SNS, with SNS stepping in as the middleman. Figure 4.6 shows an example where two functions, A and

B, interact indirectly through a bucket. The question arises: Is it also possible to send the tracing context along here so that the functions can be connected, even though a middleman is involved? The answer is generally: No.

Firstly, it is impossible to modify the payload for certain services so that the tracing context can be attached. Secondly, even if it is the case that it is possible to append the context, it is not guaranteed that it will arrive at the called function.[1]. Nevertheless, we would like to trace and analyze such workflows.

The solution lies in the use of the request contexts. The basic idea is the following: If a function makes an outbound request, an outbound context is defined and stored. If another function, which a tracing context cannot link, is executed, an inbound context is also defined and stored. If outbound and inbound contexts can be linked with the help of identifiers, then these two records can be connected. This matching of request contexts is done offline in the *Visualizer* since a global view of the records is necessary. This is because not all functions terminate in the same order as they were triggered. Thus the order of the exported records does not reflect the order of the function triggers.

Theory

Having defined the problem in the previous section and developed initial ideas on how to solve it, we would like to express how traces are constructed in the FaaS-Profiler formally.

Our goal is to order the function calls partially. A partial order is an order that arises from a relation on a set, where not every element of the set can be compared to every other element. To put it in our context, for specific function calls we can state that one must precede the other. Nevertheless, there are calls about which we cannot make such a statement.

We begin with a general definition of when a set of function calls form a partial order:

Definition 4.15 (Partial Temporal Order of Functions) Let $\mathcal{F}$ be the set of all serverless function executions. For any $f \in \mathcal{F}$ and $g \in \mathcal{F}$ with $f \neq g$, we say $f \prec g$ if f made an outgoing request at time $t_{out,f}$ that led to the execution of function g at time $t_{in,g}$ with $t_{out,f} \leq t_{in,g}$.

Note that $\mathcal{F}$ denotes the set of all function executions, not functions itself. A function can call itself during execution. FaaS-Profiler tries to establish the order between the function executions to avoid circles in the trace reconstruction. Since, as described in Section 4.1, a trace record is created

[1] An example is the asynchronous call of a AWS Lambda function. Although the tracing context was placed in the client context, it can no longer be found in the calling function.

for each function execution, ordering between the function executions is equivalent to ordering the associated records. We focus now on ordering Trace Records since FaaS-Profiler works with them as representatives of a function execution.

If a tracing context exists for a set of Trace Records, an order can be represented by the tracing contexts:

Definition 4.16 (Partial Order of Trace Records by Tracing Context) Let $\mathcal{R}$ be the set of all trace records, moreover, let $a \in \mathcal{R}$ and $b \in \mathcal{R}$ be arbitrary records with $a \neq b$. Let $\mathrm{TC_a}$ and $\mathrm{TC_b}$ be the tracing context of a and b, respectively.

We say $a \prec_{TC} b$ if $\mathrm{TC_a^{TraceID}} = \mathrm{TC_b^{TraceID}}$ and $\mathrm{TC_a^{RecordID}} = \mathrm{TC_b^{ParentID}}$

Definition 4.16 is sufficient to formally describe the first naive approach to the solution written in the problem defined above. Informally expressed, we say the function call A with trace record a happened precisely before the function call B with trace record b if both are in the same trace, i.e., for both the trace ID in the tracing context matches, and the record ID of trace record a is the parent ID of trace record b.

But as mentioned at the beginning, this definition alone is insufficient to bring all trace records into a partial order. Therefore, we present an alternative definition that orders trace records based on their associated request contexts.

Definition 4.17 (Partial Order of Trace Records by Request Contexts) Let $\mathcal{R}$ be the set of all trace records, moreover, let $a \in \mathcal{R}$ and $b \in \mathcal{R}$ be arbitrary records with $a \neq b$. Let $\mathrm{IBC_a}$ the inbound context of a and $\mathrm{OBC_a}$ the set of outbound contexts of a. We define $\mathrm{IBC_b}$ and $\mathrm{OBC_b}$ analogously for b.

We say $a \prec_{RC} b$ if $\exists \mathrm{C} \in \mathrm{OBC_a}$ such that $\mathrm{C^{Identifier}} = \mathrm{IBC_b^{Identifier}}$

In other words, if function A actuated an outgoing request whose identifiers matched the identifiers of B's incoming request, then function A happened before function B.

Based on Definition 4.16 and Definition 4.17 we can define a trace of serverless function as follows:

Definition 4.18 (Trace of Serverless Function Executions) Let $\mathcal{F}$ be the set of all serverless function executions. Then we define a partial ordering on $\mathcal{F}$ denoted by $\prec_{combined}$. For $f_1, f_2 \in \mathcal{F}$ we have:

$$f_1 \prec_{combined} f_2 \iff f_1 \text{ and } f_2 \text{ have trace records } a, b \in \mathcal{R}, a \neq b, \text{ for which Def. 4.16 or Def. 4.17 holds}$$

In the following, we use $\prec$ as $\prec_{combined}$ as introduced in Definition 4.18.

Remark One could argue to establish a complete order over the time of calling. However, this is generally not possible since it is not guaranteed that the clocks of the microservices are synchronous. For this reason, we use partial ordering based on the mappings of caller and callee.

Example 4.20 Suppose we have a set of four trace records $\mathcal{R} = \{A, B, C, D\}$, where each record belongs to an execution of a serverless function. Call A made an asynchronous call to B, which could be found in the inbound and outbound context of A and B respectively (Definition 4.17). Also, call A made a synchronous call to C, which a common trace ID could reconstruct in the tracing context and corresponding parent mapping (Definition 4.16).

According to Definition 4.18, we find two traces. One with a function calls A as root, where we know that $A \prec B$ and $A \prec C$. About the relation between B and C, we have no statement. None of the calls relates to D. Therefore D is a root of a trace with only itself as element.

Payload Injection: Forwarding the Tracing Context

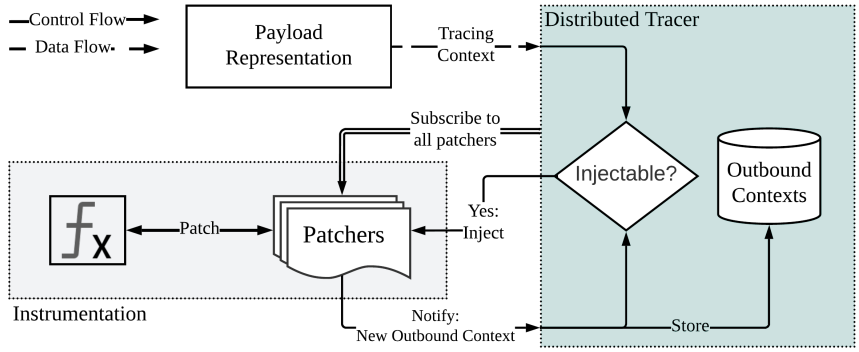


Figure 4.7: Payload Injection in FaaS-Profiler. All available patchers for libraries that can make outgoing requests are used. The tracer subscribes to these and saves each outgoing request for later export. If a payload injection is available for the outgoing method, the current tracing context of the function is injected.

FaaS-Profiler always tries to pass the tracing context if possible, so a possible triggered function learns directly from the trace and its predecessor. The *Instrumenter* accomplishes this by exploiting the patching framework and its ability to modify the payload before an outbound request is made. Figure 4.7 shows the distributed tracer and its interaction with the payload representation, Section 4.2.2, and the patching framework, Section 4.2.5. The tracer can be notified if a method could trigger another function. If the nature of the request allows adding a custom payload, the record ID and

trace ID are appended with the hope that they will be found and read by the triggered function. The actual injection of the context is highly dependent on the library used and the type of request. In Chapter 5 about a concrete Implementation in Python we provide concrete hints on how we patched specific libraries to inject payload.

Postprocessing: Connect traces afterwards

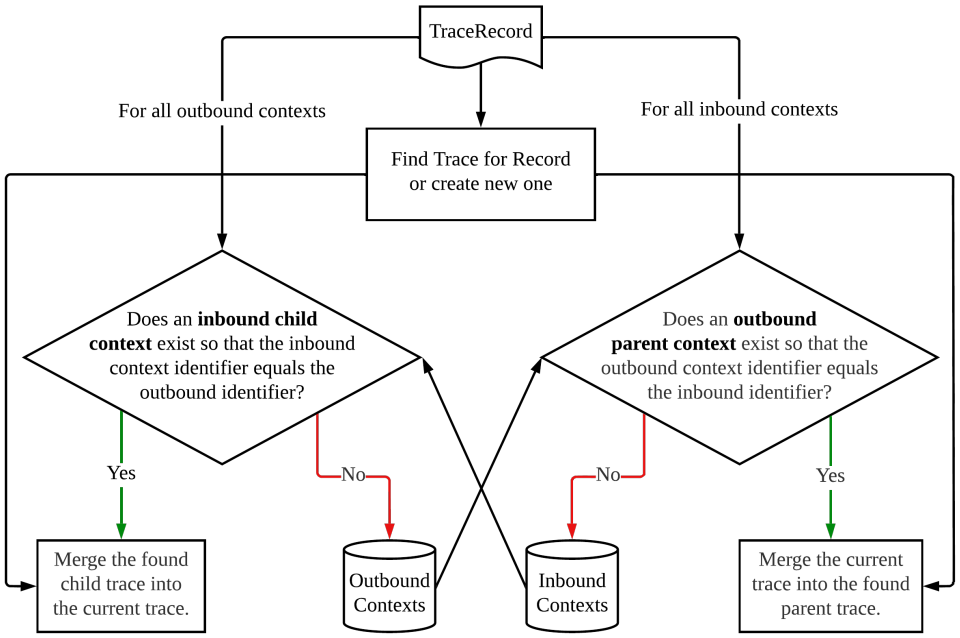


Figure 4.8: The simplified decision tree for each new *Trace Record* to subsequently merge independent traces. For each outgoing request, the record checks if a child trace has already been processed and triggered by the request. If so, the child trace is merged into the current trace. At the same time, a check is made for the incoming request context to see if a parent trace can be found that triggered this trace. If so, the trace is merged into the found parent trace. If an incoming or outgoing request context cannot be resolved, it is cached to help later arriving records progress.

Recall that not all traces can be mapped by simply forwarding the tracing context.We need an alternative way to connect parts of a trace afterwards. For this purpose, FaaS-Profileroffers a back-tracing algorithm in the *Visualizer*, i.e., it does not run at runtime of a serverless function.

The tracer exploits the request contexts for this purpose. Recall that we distinguish between inbound and outbound requests for each function execution. An *Inbound Request Context* characterizes the request that led to the execution of the currently instrumented function. *Outbound Request Contexts* characterize all outbound requests to other services undertaken during

execution.

Section 4.1.3 shows that we associate a set of identifiers with each Request Context. These should help to find incoming and outgoing requests in the postprocessing step. For the role of the identifiers, we refer to Example 4.21.

Example 4.21 (Connecting AWS Lambda calls by identifier) Assume a Lambda function A triggers a Lambda function B asynchronously. After executing the trigger for function B, AWS returns a request ID that is the same as the one that function B receives when it runs. Therefore, for function A, we store the request ID and function name as the outbound request context identifier, and once function B is triggered, we extract its function name and request ID. Once both functions export their trace record, we find that the records have matching identifiers in their outbound and inbound contexts. We can thus connect both function calls.

In Figure 4.8, we show the simplified decision tree of the postprocessing algorithm. The basic idea is the following: For each trace record, we handle the stored inbound and outbound context detected and exported by the *Instrumenter*. We look to see if we have already seen a trace record that made an outbound request whose identifiers match the inbound request context of the currently handled record. If so, we have found a parent record and a new trace in which we append the record. If not, we store the record temporarily with the hope of being resolved later. We do the same process for each outgoing request context of the trace record, only this time, we don't look for a parent but for possible children that can be appended.

In addition to the check for matching identifiers, the temporal order is also checked. This means that only an outbound and inbound request will be connected if the outbound request was made before the inbound request. To account for possible clock differences between actors, a tolerance is added.

Algorithm 1 shows the pseudo-code of the backtracing algorithm. The algorithm can be applied manually for a batch of new records or each new record. The latter allows live updating of traces, but requires a database for recording the request contexts.

The algorithm was designed to know that trace records can be processed in any order. So the algorithm behaves the same for any order of records. The concept of ”helping others to make progress, ” used in multiprocessor operation, was applied. It is elementary that we cache request identifiers that are not directly resolvable, hoping that a later processed record will find itself as parent or child.

Algorithm 1 Offline Backtracing Algorithm

```
Traces ← ∅
OutboundRequests ← ∅
InboundRequests ← ∅
procedure MergeTraces(UnprocessedRecords)
    for each record ∈ UnprocessedRecords do
        trace ← find or create new trace for record
        if TracingContext_record has no parent ID then
            ResolveInboundContext(InboundContext_record)
        end if
        for each OutboundContext ∈ OutboundContexts_record do
            ResolveOutboundContext(OutboundContext)
        end for
    end for
end procedure

procedure ResolveInboundContext(InboundContext)
    if ∃ ParentRequest ∈ OutboundRequests
    s.t. InboundContext^Identifier = ParentRequest^Identifier then
        ParentTrace ← Trace of ParentRequest
        Merge Trace into ParentTrace
    else
        Store InboundContext → InboundContexts
    end if
end procedure

procedure ResolveOutboundContext(OutboundContext)
    if ∃ ChildRequest ∈ InboundContexts
    s.t. OutboundContext^Identifier = ChildRequest^Identifier then
        ChildTrace ← Trace of ChildRequest
        Merge ChildTrace into Trace
    else
        Store OutboundContext → OutboundContexts
    end if
end procedure
```

Chapter 5

Implementation

In this chapter, we will dive deeper into implementing the FaaS-Profiler. While chapter 4 was more about the high-level design of the tool, we will now provide more details about a concrete implementation. Recall that the FaaS-Profiler is divided into two main components: the *Instrumenter* [1], installed with the function to be examined, and the *Visualizer* [2], installed locally and used to explore the results.

In the following, we focus on a realization in Python applied to the serverless functions deployed on Google Cloud Platform and Amazon Web Services.

5.1 Instrumenter for Python

The following subsections are roughly structured according to the *Instrumenter's* control flow. We will first cover how the *Payload* of the functions is resolved. Then we will discuss how the *Patchers* are implemented in Python. In the end, we examined existing *Measurement*, *Capture*, and *Exporter* routines.

5.1.1 Instrumentation & Configuration

Listing 12 Instrumentation of a serverless function with handler

```
import faas_profiler_python as fp

@fp.profile()
def serverless_handler(*args, **kwargs):
    pass
```

[1] pip-installable package under the name `faas-profiler-python`
[2] pip-installable package under the name `faas-profiler`

To instrument, a serverless function with the FaaS-Profiler in Python, only one line of code needs to be added. Listing 12 shows an example. Note that only the handler, i.e., the function entry point, needs to be edited accordingly.

The entry point for the *Instrumenter* is a special function that takes the serverless handler's method reference and associated arguments as parameters. The handler is not executed until the *Instrumenter* commands it. This provides the ability to execute logic before and after the serverless function is executed. Also, operations can be performed based on the results of the instrumented function.

The *Instrumenter* uses a Python's decorator `fp.profile` to instrument the function in a user-friendly way. Decorators are special functions that take a function with its arguments as parameter and return a function themselves [9]. The profiler's decorator returns the instrumented handler instead of the original handler.

Configuration

Listing 13 Example configuration for profiling a function

```
measurements:
  - name: network::Connections
  - name: network::IOCounters
  - name: memory::LineUsage
  - name: memory::Usage
  - name: cpu::UsageOverTime
  - name: information::Environment
  - name: information::OperatingSystem
  - name: disk::IOCounters

captures:
  - name: aws::S3Capture
  - name: aws::EFSCapture
    parameters:
      mounting_point: /efs/mount

exporters:
  - name: common::Console
  - name: storage::AWSVisualizerUploader
    parameters:
      region_name: eu-central-1
      bucket_name: faas-profiler-records
```

The client library can be configured individually for each serverless function.

For this purpose, a configuration file can be passed in YAML format. The following section briefly describes the configuration options using a YAML file, as this is the most descriptive. Note that the configuration via the file is one possibility of many. Other ways, like via environment variables or program arguments, can be used too. We refer to Appendix A.2 for further guidance on using environment variables.

The main task of the configuration is to tell the profiling tool which measurements, captures, and exporters to use. This is done by simply defining defines a list for each of these three components, which specifies to the tool which routines are to be loaded and executed. Each routine is assigned a unique name. This is used once to import the routine dynamically and later when exporting it as identification in the record data.

An example configuration is shown in Listing 13; Section 5.1.4, 5.1.5, and 5.1.6 will go into the measurement, capture and exporter routines shown.

5.1.2 Payload

In the design chapter, Chapter 4, we have defined that the *Payload* is defined by all data available to the function at the start of execution. Once a serverless function is instrumented, the client breaks down the payload to get more information about a possible tracing context and the inbound context. Since the payload differs significantly from provider to provider, a separate representation is defined for each. In the following sections, we provide details on the specific payload representation for AWS Lambda and Google Cloud Functions.

Payload Representation for AWS Lambda

Based on the official documentation of AWS Lambda, a Lambda handler is passed two arguments when called: An *Event* object as the first argument and a *Context* object as the second argument.

The *Event* object contains information about the service that triggered this function.

The *Context* object contains information about the function call, its runtime, and the Lambda function itself.

Recall that each payload representation, Section 4.2.2, in the FaaS-Profiler has two tasks: Detect the *Inbound Context* and extract the *Tracing Context*.

Determination of the Inbound Request Context Based on the *Event* argument for the AWS call, the FaaS-Profiler can create the inbound request context. As described in the Background Chapter 2, Lambda functions can generally

be triggered by several other AWS services. Event-driven services [3] send the lambda functions a list of records representing items to be processed by the function in the context of the triggering service. In the case of DynamoDB, for example, these are new entries or, in the case of S3, new uploaded objects. FaaS-Profiler extracts from each of these the AWS service and operation, as well as identifiers, to help the backtracing algorithm identify the request of an outgoing service.

General calls, either by the gateway, SDK, or StepFunctions, can be recognized by the missing records; the event parameter carries the available payload of the function.

Determination of the Tracing Context Extracting the tracing context can be seen as the reverse operations of payload injection. The payload resolver tries to find the context in the headers and payload of the request independently of the event. The location in the payload where the context was placed varies significantly from event to event. For example, in the case of messages for SQS or SNS, it was sent in the attributes of the message; in the case of a call to AWS Lambda, it is sent in the client context.

Payload Representation for GCP Functions

The payload for the GCP function can be roughly divided into HTTP payload and event-driven payload.

In the case of an HTTP request, a representation of the HTTP request is passed to the function, which makes the headers and data available.

If the function is triggered due to an event in another service, an *Event* and *Context* object are passed to the function as an argument. Where *Event* gives information about the triggering data element, e.g., a queue message or a cloud storage object. The *Context* object describes the event in more detail, e.g., specifying the event ID, cloud storage bucket, or the message queue's name.

Determination of the Inbound Request Context The payload representation for Google Cloud functions detects whether the request is an HTTP request or an event is the origin of the execution. In the case of event execution, the event and context object is used to detect the Google Cloud service and operation. Furthermore, properties like event ID, queue name, or bucket name are read to find the inbound request identifier.

[3] Such as AWS Cloud Front, AWS CloudWatch, AWS DynamoDB, AWS S3, AWS SNS, AWS SQS, AWS SES, AWS Code Commit, and AWS Kinesis

Determination of the Tracing Context Similar to AWS requests, the location of the tracing context is highly dependent on the request type and the event reason. Again, the request headers and payload are used to find the tracing context. In the case of queue messages, the context may be in the meta-data.

5.1.3 Patchers

The Python version of the FaaS-Profiler implements the patching framework presented in the design chapter (Section 4.2.5). The client uses the open-source package `wrapt` by Graham Dumpleton et al., which wraps arbitrary functions with a wrapper function [6]. The tool was developed with a focus on transparency, correctness, and low overhead and thus ideally fits the characteristics required in Section 4.2.5.

The client has several patchers provided out of the box. In the following, we would like to discuss the patching strategy of the Cloud Provider SDKs.

AWS SDK for Python

The AWS SDK for Python, called Boto3, lets you create, configure, and manage services in an object-oriented way [58]. For example, the library can call a Lambda function, upload files to S3 or create a new entry in DynamoDB. Boto3 is automatically installed with all Lambda functions.

The Python client of the FaaS-Profiler defines a patcher for `botocore`, on which Boto3 is based, and provides a low-level interface to AWS services [59].

It takes advantage of the fact that in Boto3, each client for all services in AWS inherits from a basis client. When performing an outbound request, it invokes the same method, which is excellent as a target method for the function patcher. Based on the client and the arguments, the requested AWS service and the operation to be performed can be inferred. Also, using the arguments and the service's return values, the request's identifiers can be inferred.

GCP SDK for Python

Unlike the Python SDK for AWS, Google Cloud does not have one SDK that can be used to request all services. Google divides the kit into different packages, each of which controls one service. No inheritance hierarchy could be found between the packages, so FaaS-Profiler defines a separate patcher for each library[4]. Since there is a patcher for each service library, determining the service and operation of the outgoing request is trivial.

[4]There are patchers for Pub/Sub, Cloud Functions, Cloud Tasks, and Cloud Storage.

Again, the function arguments and function results are used to determine the identifiers.

5.1.4 Measurements

The current version contains some predefined measurement routines, which can be activated in the configuration if required.

All following the interface in Listing 7. Recall that we distinguished between *periodic* and *simple* measurements. Periodic measurements collect more points over a metric during the execution of the function at a user-defined interval. Simple measurements measure at the beginning and the end of the function.

FaaS-Profiler provides measurements to determine CPU, memory, disk, and network consumption during the execution of the instrumented function. There is also the possibility to read information about the runtime and the system.

Most measurements are based on the Python package `psutil`, which we will briefly introduce in the following. After that, the working measurement routines are presented.

The Package `psutil`

`psutil` is an open-source Python package developed by Giampaolo Rodola et al [7]. It provides a Python interface to receive information about processes and systems. The interface was designed as a cross-platform tool and supports most common platforms through a common API. As we will see in the following sections, in the case of a Linux system, the `/proc` filesystem is used to get information about the running process. The process filesystem, `procfs` for short, is a virtual filesystem. Under most Unix-like systems, it is dynamically generated by the kernel and provides data and process parameters [3].

Memory Consumption

The FaaS-Profiler contains two different measurement routines to measure the memory consumption of a function. The routine named `memory::LineUsage` gives the current total memory usage for each line of the source code and the difference to the last line. In contrast, `memory::Usage` provides a time-dependent measurement that reflects the memory usage throughout function execution.

Both approaches use `psutil`, to fetch memory information for a process. The proc file `proc/pid/statm` is read out, which provides information about the memory usage measured in pages. We use the value resident set size (RSS), which is the part of the memory that the process occupies in RAM [61].

Memory Consumption per Line The idea to implement how to measure the memory usage per line was taken from the Python package `memory-profiler` [55] by Fabian Pedregosa et al. The approach uses Python's ability to set custom code trace methods, which are called on various events during code execution, for example, when a block or a new line is executed or a block returns [10]. The line profiler takes advantage of this to be notified if a new line is executed. If this happens, the new line is recorded, and the memory usage is recalculated using the method described above, from which the difference from the previous line can be calculated. A fine-grained reconstruction of the memory usage is created.

Memory Consumption over Time The measurement routine `memory::Usage` is a periodic routine that queries and records the memory usage of the process in which the instrumented method runs at one millisecond intervals. The method described above with `psutil` is used for the query.

CPU Consumption

The client provides a periodic measurement named `cpu::Usage` to measure the CPU usage in percent over the time of function execution. Recall that periodic measurements are executed in a child process and get as an argument the process ID of the main process in which the instrumented serverless function is executed.

`psutil` is used to read the proc file `/proc/pid/stat`, which outputs various values for the process with the given ID. It contains the values `utime` and `stime`, which represent the CPU's time in user and kernel modes, respectively [50]. To determine the percentage of consumption, we compares the CPU time with the last query to calculate the percentage difference.

The measurement routine `cpu::Usage` now uses the method `measure`, which is called in a time interval of one millisecond, to record the CPU consumption in the way written above.

Disk Consumption

The measuring routine `disk:IOCounters` allows monitoring of the disk I/O counters during the execution of the function. The routine works according to the snapshot principle. Before the immediate start of the instrumented method, the current I/O counters plus the number of read and written bytes are read out. The same is done after the end of the execution so that the difference between the start and end snapshots can be returned.

To read the I/O counters of a process, we read the proc file `/proc/pid/io`, which contains the values `syscr`, `syscw`, `read_bytes` and `write_bytes` for the given process. `syscr` and `syscw` specifies the number of I/O write and

read accesses respectively. `read_bytes` and `write_bytes` gives the number of bytes read and written respectively [50].

Network Connections and Consumption

FaaS-Profiler provides two measurement routines to record network I/O counters and network connections, respectively.

Network Connections To record the network connections made during the function's execution, the client provides the periodic measurement routine `network::Connections`. This uses `psutil`'s support to read the `/proc/net` folder. This contains different files to read information about the network layer [50]. The function reads the socket table for the different network protocols to get a list of currently open and active connections. Since this list contains only open connections, the client implements this measurement as periodic, so this list is updated intermittently. To avoid recording connections twice, the routine uses the socket descriptor to detect if a connection has occurred. This is feasible because the socket descriptor under Linux is equivalent to file descriptors, which are unique per process.

In the end, the measurement returns a list of performed connections. Each connection is described by the remote address, connection type, and number of connections to one remote address.

Remark This measurement has its limitations. The `/proc/net` system returns only the active connections. Therefore the polling delay of the periodic measurement must be small to reflect all connections of the function. We will see in Chapter 6 about the evaluation that we can catch most of the connections with a delay of one millisecond. We, therefore, point out that the measurement is not perfect.

Network I/O Counters Similar to the I/O counters for the hard disk, the measuring routine `network::IOCounters` also works according to the snapshot principle. It used the proc file `proc/net/dev` to get network device status information. Readable is the number of sent and received bytes and packets. Also, the number of errors during send and receive and the number of incoming and outgoing dropped packets are returned [50].

The measuring routine makes a snapshot at the instrumented function's beginning and after the execution's end. The difference between these records is returned and thus represents the network I/O counters during the execution of the function.

System & Runtime Information

FaaS-Profiler also includes measurements to gather information about the system and the runtime. For example, the operating system, Python compiler, or CPU architecture can be read out. Furthermore, a measurement determines whether a container is warm and, if so, for how long. For this, a file is stored in the container. If this exists at a subsequent call of the function, then the container is warm. The creation time of the indicator file indicates how long the container was kept warm.

Summary

All measurements described here have been shown to work with AWS Lambda and Google Cloud Functions.

In addition to reading performance metrics via the `procfs` file system, we also tried to read hardware counters via PAPI. PAPI [1], short for Performance Application Programming Interface is a tool to read the performance hardware counters of a microprocessor. In general, PAPI allows more precise and concrete metrics, such as the number of executed instructions. It was not possible to retrieve these values on both platforms. We assume that the MircoVM in which the function was executed does not have the rights to query these hardware counters or does not support PAPI.

5.1.5 Captures

As the design chapter highlights, captures record calls to specific functions. FaaS-Profiler currently has two specific capture routines for AWS: one that records every action to an EFS file system and one that records every operation to AWS cloud storage S3. The capture results can be viewed as cloud-specific performance metrics so that analysis can be undertaken on time spent on S3 or EFS.

In the Python implementation, the recording of the calls is realized by patching.

EFS Access

Recall from the background chapter that a file system can be attached to an AWS Lambda function to provide persistent storage across function calls. The NFS (Network File System) based AWS service Elastic File System, EFS for short, offers the possibility to mount a flexibly scalable file system through a local mount point to the runtime of the Lambda function [60]. For example, if the file system is mounted at the `/mnt/efs` point, the Lambda function can interact with the system through normal I/O operations. The mount point is a configuration parameter before the function is deployed. This means that

the mount point is fixed and known before execution for each execution of the current version of the function. The capture routine uses this property for EFS.

Since the filesystem interacts with normal I/O functions and not with an SDK, a patcher for Python's built-in function `open` was defined. The `open` function is used to open data, whether it is being read or written [8]. By patching, we are notified if someone performs file access. By filtering the accesses that go to an EFS mount point, we can thus record EFS accesses. Besides information about file name, type and size, it also records how long the access takes. The capture routine `aws::EFSCapture` exports a list of file accesses to specific EFS mount points.

S3 Access

To record operations with the AWS S3 service, the AWS Python SDK Boto3 is patched, see Section 5.1.3. Therefore, we registered the S3 capture routine `aws::S3Capture` as an observer of the botocore patcher, so we will be notified if an operation is performed. After filtering non-S3 operations, we can export a list of S3 operations and record which objects were accessed and how long the process took.

5.1.6 Exporters

The Python client includes three simple export routines and supports formatting the record data in JSON and YAML. JSON is preferred, and additional formatting can be easily added. The exporter `common::Console` outputs the data in the standard output. The upload exporters `common::AWSVisualizerUploader` and `common::GCPVisualizerUploader` upload the data to the respective cloud storage of the provider. For this purpose, the FaaS-Profiler provides a standard interface, *Record Storage*, so the offline tool *Visualiser* can reliably find it. Both exporters must be passed a bucket for uploading, to which the function has access.

5.2 Visualizer and CLI

The *Visualizer* offers the user the possibility to visualize and analyze data collected via the client. Furthermore, it offers a rudimentary command line interface to deploy and profile functions. Application notes can be found in Appendix A.3. In the following, we will briefly discuss the *Visualizer*.

One of the main tasks is to post-process the record data uploaded through the *Record Storage* interface. For this purpose, the package implements the algorithm presented in Section 4.2.6 to establish the partial order between records. During post-processing, the algorithm also combines multiple traces into

Profiles. *Profiles* represent the repeated invocation of a particular serverless function and are intended to allow analysis across different requests.

FaaS-Profiler offers a Dash-based web application that can be run locally for visualization. Dash is an open-source Python package for easy visualization and analysis of data [49]. It is based on the web framework Flask, the charting library Plotly and the interface framework React. It is provided free of charge by the company Plotly.

The visualizer dashboard has a graph algorithm to represent the reconstructed partial order. The graph shows which function made which outgoing requests and which function is executed as a result. The latency between the execution of the outgoing request and the start of the triggered function is calculated and visualized.

Besides the graph, the dashboard has analyzer routines. These can subscribe to specific data in the records, analyze them and subsequently output a chart with the help of Plotly. Analyzers have been defined for all measurements and captures presented above.

In Chapter 6 case studies, the visualizer can be seen in action.

Chapter 6

Evaluation

In this chapter, we will present and evaluate the results of the Python implementation of the FaaS-Profiler applied to serverless functions and applications. As in the previous Chapter 5 about the Python implementation, we focused our evaluation on functions deployed on the Google Cloud Platform and Amazon Web Services. We provided the source code and serverless deployment configuration to a repository. Details of the source code can be found in Appendix A.1.

In the first Section 6.1 we evaluated the tool using smaller micro-benchmarks to determine the upcoming overhead of certain parts of the profiler. In the second Section 6.2 we present longer-running and more complex serverless applications and show how the distributed tracer of FaaS-Profiler reconstructs these workflows.

6.1 Micro-Benchmarks

To examine the FaaS-Profiler in terms of overhead, we tested it with simple functions. All functions were run with Cloud Functions from Google and Lambda from Amazon. After presenting our methodology for the following time and memory measurements in Section 6.1.1, we start Section 6.1.2 with a *No-op* function to determine the pure overhead of the instrumenter without active measurements and tracing. Section 6.1.3 evaluates the cost incurred when the FaaS-Profiler actively intercepts and records outgoing requests. Section 6.1.4 analyzes the additional cost of collecting function metrics during the function run.

6.1.1 Methodology

For AWS Lambda, we chose the region `eu-central-1` in Frankfurt. We gave each function 1024 MB of maximum memory and a timeout of 10 seconds.

The FaaS-Profiler was statically added as a Lambda layer to each function. AWS Lambda reports the maximum memory used in megabytes, the exact execution time in milliseconds after each function execution, and the time to initialize the container if it was the first call to the function. We collected all these values for our evaluation and performed our analysis based on them.

We used the `europe-west3` region in Frankfurt for Google Cloud functions. Again, we gave each function a maximum RAM of 1024 MB and a timeout of 10 seconds. FaaS-Profiler was specified as code dependency directly and, as explained in Background, automatically installed and compiled during deployment. Google reports the execution time in milliseconds after each function execution. We collected this data for time validation. Unlike Amazon, Cloud Functions does not return the maximum used memory after each execution. Instead, an average value for the consumed memory can be queried and sampled in 60-second intervals[1]. We used these average values to analyze memory behavior.

All functions were executed on the `x86_64` architecture.

Each function was performed 100 times in a row on a cold container. The overhead for the cold start was only experienced on the first call.

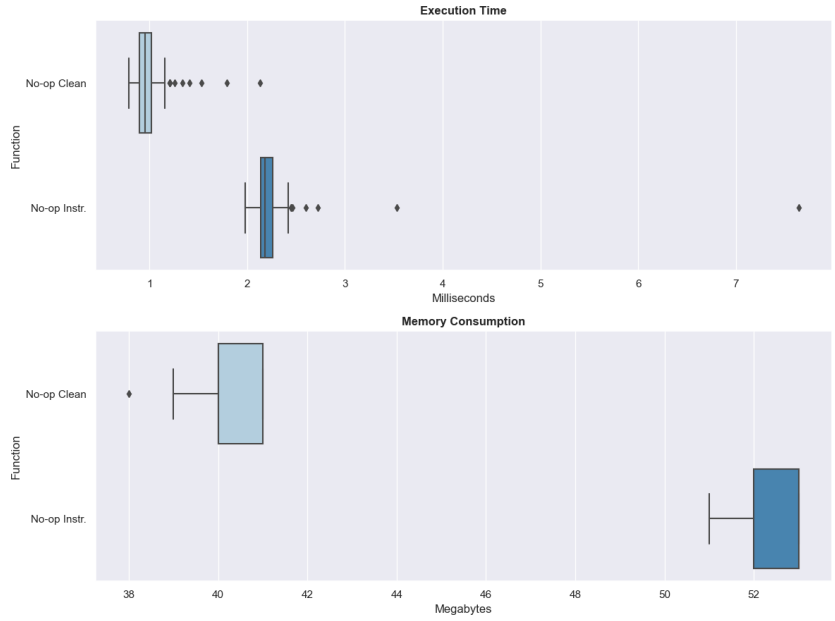


Figure 6.1: Memory consumption and execution time of the *No-op* function with and without FaaS-Profiler for AWS Lambda. Memory consumption increased by 30.63% and execution time by 129% on average.

[1]Information about Function metrics: https://cloud.google.com/monitoring/api/metrics_gcp#gcp-cloudfunctions

6.1.2 No-op Function

To obtain a initial baseline for the following measurements, we first performed an *No-op* function without and with FaaS-Profiler. *No-op* is an empty function that does not perform any calculations and returns immediately. The profiler was run in ”idle” mode, so no measurement captures or tracers were activated. The goal is only to show how much overhead is created when the tool does nothing but resolve the payload and initiate a trace record at the end. The trace record was created but not exported.

AWS Lambda Results

Figure 6.1 shows the results of the *No-op* function for AWS Lambda after 100 runs. FaaS-Profiler raises the mean memory consumption by 30.63% on an Empty function from 40.32 MB to 52.67 MB. The mean execution time recorded an increase of 129% from 0.996 ms to 2.284 ms.

Google Cloud Functions Results

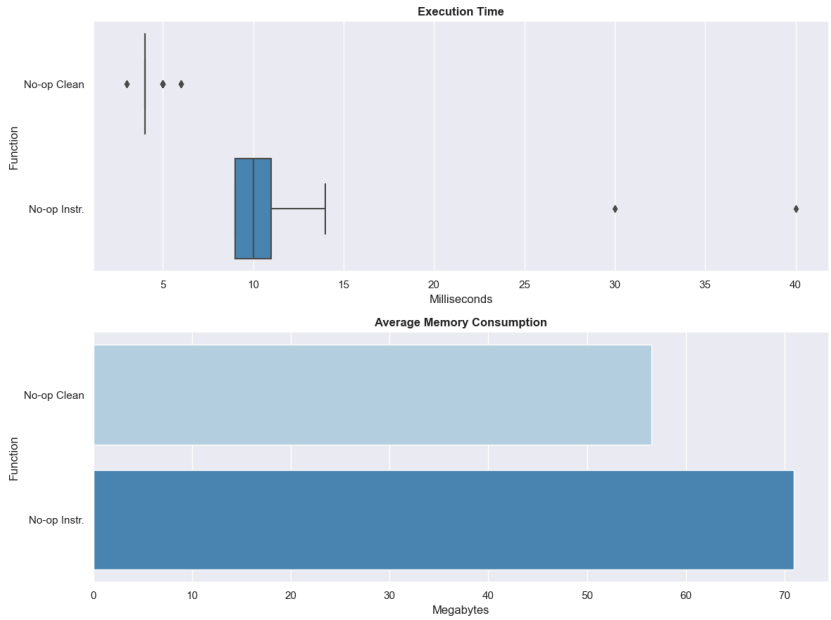


Figure 6.2: Memory consumption and execution time of the *No-op* function with and without FaaS-Profiler for Google Cloud Functions. Memory consumption increased by 25.53% and execution time by 96.86% on average.

Figure 6.2 visualizes the results after 100 runs of the *No-op* method for Google Cloud. On average, the execution time increases by 96.86%, from 5.87 ms to 11.56 ms. Memory consumption increases by 25.53%, from 56.52 MB to 70.94 MB.

6.1.3 Functions with Tracing

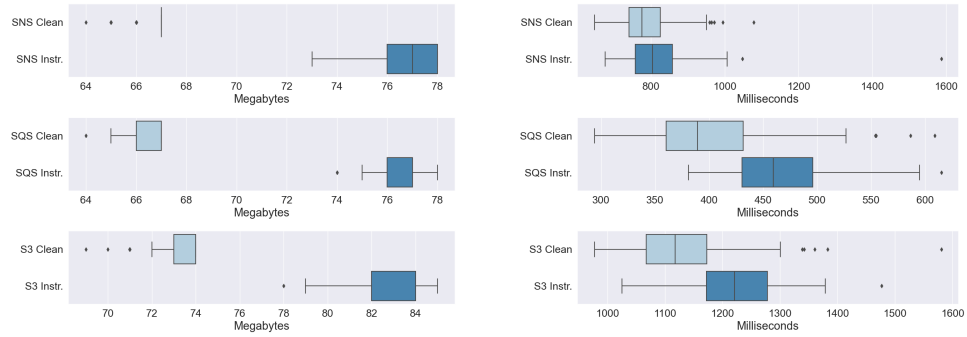
Function	Description
Storage Write	50 consecutive writes to a cloud bucket.
Message Publish	50 consecutive publications of messages.
Queue Publish	50 consecutive publications of tasks to a queue.

Table 6.1: Micro-benchmarks to evaluate tracing overhead in FaaS-Profiler.

In the following, we will evaluate simple functions that make outbound requests captured, evaluated, and exported by the FaaS-Profiler using patching. The functions are intended to show the overhead that the tracer and the patching framework incur in terms of additional execution time and memory consumption.

We present three functions: *Cloud Storage Write*, *Message Publishing*, and *Task Publishing*, summarized in Table 6.1. In the first case, the function uploads a small file (18 B) 50 times to a cloud storage bucket. The messaging function publishes a message 50 times to a message queue. The task function queues 50 jobs into a job queue. The S3, SNS, and SQS services were used for the AWS Lambda function. We used the Cloud Storage, Pub/Sub, and Cloud Tasks services for the Google Cloud function. Again, all functions were executed 100 times in a row.

FaaS-Profiler successfully intercepts and records all 50 outgoing requests and, if possible, modifies the request payload with the current trace information. A trace record was created for each function execution and exported via a cloud bucket.



(a) Memory consumption increased by 13.55% and 15.5%.

(b) Execution time increased by 3.41% and 15.5%.

Figure 6.3: Overhead in memory consumption and execution time for functions with active patching in AWS Lambda.

AWS Lambda Results

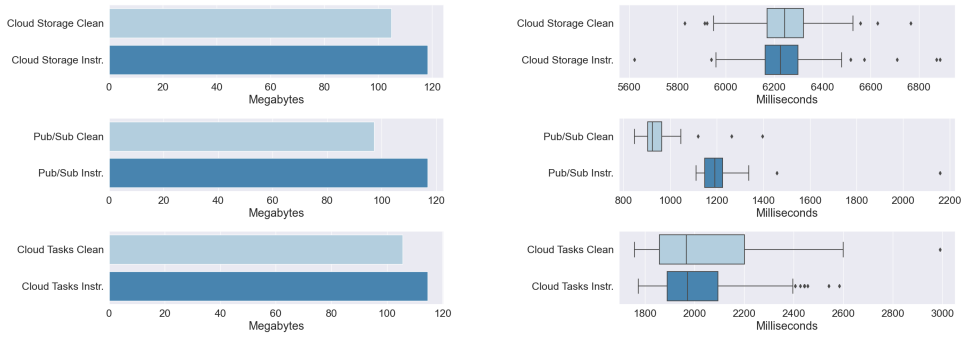
Figure 6.3 shows the overhead caused by FaaS-Profiler with active patching for AWS Lambda.

For SNS messages, memory consumption increased by 15.37%, from an average of 66.67 MB to 76.92 MB. On average, execution times were raised by 3.41%, from 792.05 ms to 819.08 ms.

In the case of SQS publications, both memory consumption and execution time increased by approximately 15.5%, from 65.30 MB to 76.68 MB and 404.00 ms to 466.82 ms on average, respectively.

Tracing the S3 uploads resulted in a memory increase of 13.55% and an execution time of 8.02%, respectively. On average, memory usage increased from 73.2 MB to 83.12 MB, and execution time from 1132.74 ms to 1223.53 ms.

Based on the micro-benchmarks, intercepting and tracing outbound requests from AWS results in a mean increase in memory consumption of 13.55% - 15.5% and an increase in compute time of 3.41% - 15.5%. The cost per GB-second increased by 25.5% on average.



(a) Memory consumption increased by 8.53% and 20.26%.

(b) The execution time increased by a maximum of 24.7% on average

Figure 6.4: Overhead in memory consumption and execution time for functions with active patching in Google Cloud Functions.

Google Cloud Functions Results

Figure 6.4 shows the overhead caused by FaaS-Profiler with active patching for Google Cloud Functions.

Catching and storing 50 Pub/Sub messages increased memory usage by 20.26%, from 97.24 MB to 116.94 MB. The execution time increased on average by 24.7%, from 988.83 ms to 1233.24 ms.

For Cloud Tasks, we recorded an increase of 8.53% in memory consumption from 105.69 MB to 114.72 MB. No growth was noted in the execution time. On the contrary, the time with instrumentation was 2028.64 ms and without 2058.25 ms. The number of active instances at 100 invocations was one for both measurements.

Tracing the upload of the cloud storage files increased the memory consumption from 104.97 MB to 118.47 MB by 12.87%. Similar to the cloud task, the execution time did not increase. With instrumentation, it averages 6239.93 ms and without 6275.97 ms Again, the number of active instances is the same for both experiments. Google reported two active instances during execution.

Based on the micro-benchmarks, intercepting and tracing outbound requests from GCP results in a mean increase in memory consumption of 8.53% - 20.26%. The execution time increased by a maximum of 24.7% on average. The cost per GB-second increased by 24.6% on average.

6.1.4 Functions with Profiling

Function	Description
Website Download	Download HTML content from `example.org`.
Path Finding	Path finding with obstacles using a potential field.
S3 to EFS	Load images from a bucket to EFS (only on AWS).

Table 6.2: Micro-benchmarks to evaluate profiling in FaaS-Profiler.

We test FaaS-Profilers on various functions below to determine the overhead incurred by collecting metrics. We will first briefly introduce what operations the function undertakes and what we want to measure with it. Then we present the results on Google Cloud and Amazon Web Services and show the induced overhead.

The *Website Download* function makes an HTTP request to `example.org` to download the HTML content. It then calculates and returns the size of the content in megabytes. The profiler was set for the function of determining all network connections and network I/O counters. The website has a size of HTML content of 1305 bytes (1,305 KB), in addition to overhead caused by the headers and establishment of the HTTP connection. At the time of the measurements[2], a DNS lookup, for `example.org`, gave an A record on `93.184.216.34` (ipv4) and an AAAA record on `2606:2800:220:1:248:1893:25c8:1946` (ipv6). The website was made via an HTTP request responding on port 80.

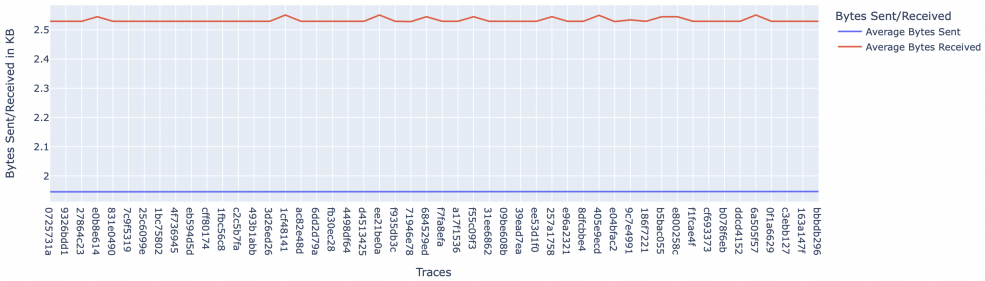
[2] 01 October 2022

Path Finding is a serverless function to find a collision-free path using a potential field. The function code was written by Atsushi Sakai et al. [56] and adapted to the serverless environment. The function was implemented with the Python library NumPy, which aims to measure memory and CPU consumption with the FaaS-Profiler.

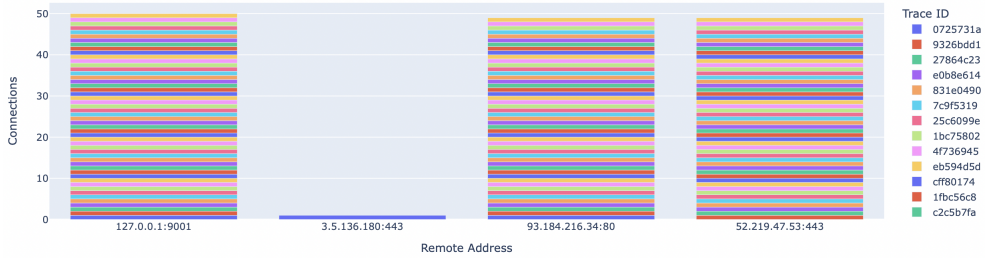
Last, we use the *S3 to EFS*[3] function (only on AWS) to evaluate the capture functionality. The function downloads images from a bucket and stores them in an attached EFS store. FaaS-Profiler measures the time and memory of S3 and EFS requests.

For all functions, trace records were exported to AWS S3 and Google Cloud Storage, respectively.

AWS Lambda Results



(a) Recorded average sent and received bytes.



(b) Recorded network connections.

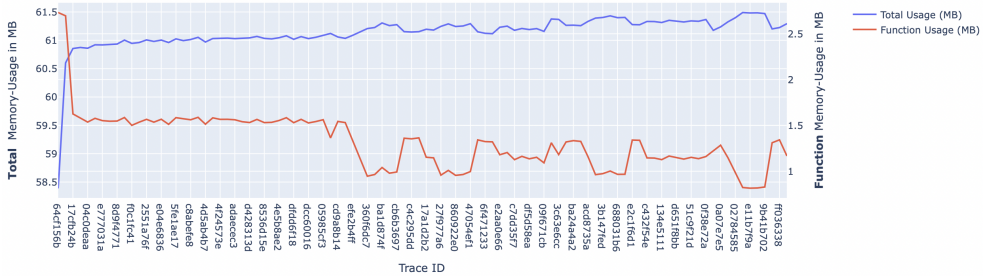
Figure 6.5: FaaS-Profiler visualization of network I/O and connection for *Website Download* on AWS Lambda.

Website Download Figure 6.5 shows the results of the FaaS-Profiler for the *Website Download* function. The function was called 50 times. A stable rate of bytes sent and received can be seen. We also measured the network I/O to verify the results counters of the *No-op* method. A constant rate of 1.3 kB sent

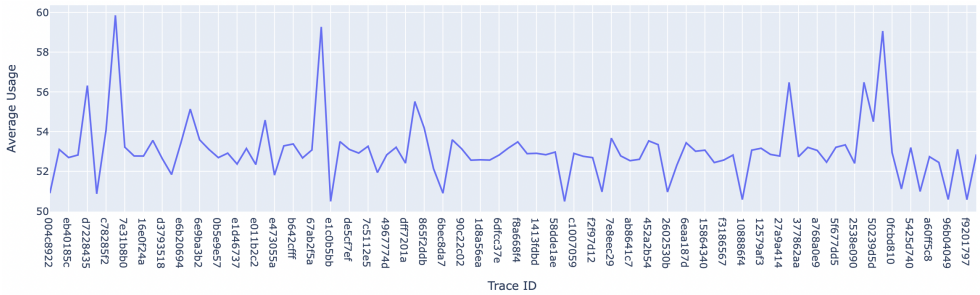
[3]Elastic File System (EFS) is a network-attached storage that can be configured for Lambda functions.

bytes, and 0.45 kB received bytes was recorded. FaaS-Profiler thus correctly records the more bytes consumed in network access. In terms of network connections, it should be noted that AWS Lambda makes connections to the local IP addresses 127.0.0.*X* and 169.245.*X*.*X* every time it is called, both in the *No-op* function and in the *Website Download* function. In addition, HTTPS requests to S3 (52.129.47.53) were captured even though no explicit S3 requests were made. We assume that importing the AWS Cloud SDK (`boto3`) and creating the S3 client triggers a request. The FaaS-Profiler later used this client to export the data, but only after it stopped recording connections. Also, the IP address of `example.org` (93.184.216.24) was recorded in majority of the HTTP calls.

Collecting I/O counters and network connections for AWS Lambda increased average memory consumption by 59%, from 44.86 MB to 71.3 MB. Execution time increased by 10.37%, from 359.41 ms to 396.67 ms. The cost per GB-second increased by 75.49% on average.



(a) Recorded average memory consumption. Blue shows the total consumption and red the consumption only by the function itself. The sampling rate was $0.001s$



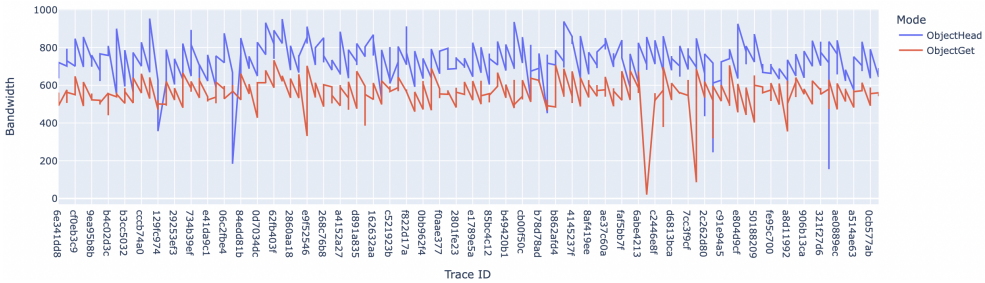
(b) Recorded average CPU consumption in percentage. The sampling rate was $0.1s$

Figure 6.6: FaaS-Profiler visualization of memory and CPU consumption for *Path Finding* on AWS Lambda.

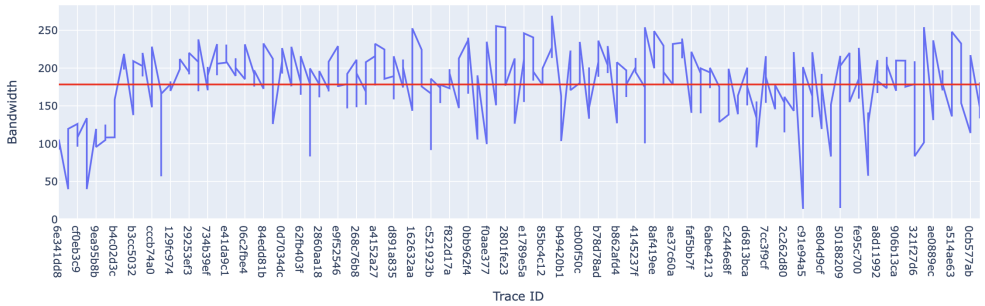
Path Finding Figure 6.6 shows the memory and CPU consumption of the *Path Finding* function of the 100 calls ordered by start time. Note that the red line shows the consumption of the function itself; previous consumption

has been subtracted, while the blue line shows the total consumped memory. Only the consumption of the process in which the instrumented method is executed is monitored.

Collecting this information increased memory consumption by 37.0% from 75.33 MB to 103.21 MB and execution time by 3.9% from 2181.09 ms to 2266.31 ms. The cost per GB-second increased by 42.36% on average.



(a) Recorded bandwidth for Get and Head operation for S3 in Mbps.

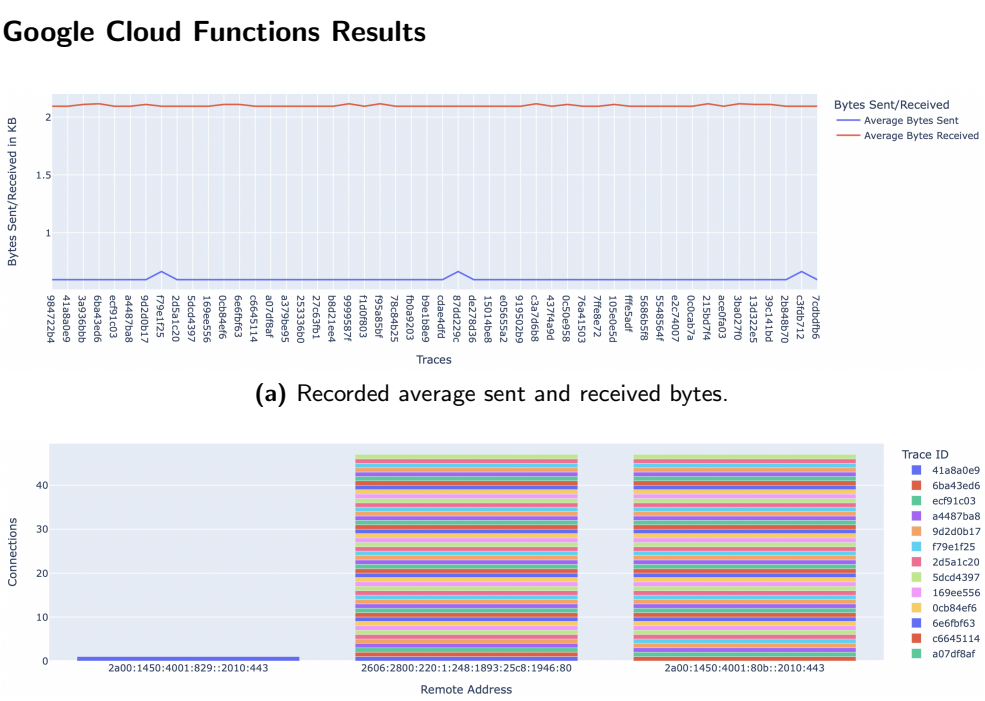


(b) Recorded bandwidth for EFS write accesses in Mbps.

Figure 6.7: Visualization of the FAAS-PROFILER for EFS bandwidth and S3 Access performance through the *S3-to-EFS* function.

S3 to EFS Using the *S3 to EFS* function, we captured cloud-specific metrics for AWS. Figure 6.7 shows the bandwidth in Mbps for write accesses to EFS and read accesses for S3. For writes to EFS, we determined a mean bandwidth of 171.76 Mbit/s after 100 executions of the function. For API reads to S3, we recorded a mean bandwidth of 647.84 Mbit/s. Note that get requests in S3 are split into a head request executed first, followed by a get request.

Recording the cloud performance metrics produced a 12.37% from 430.06 ms to 483.28 ms increase in execution time and a 6.80% from 269.02 MB to 287.32 MB increase in memory usage. The cost per GB-second increased by 20% on average.



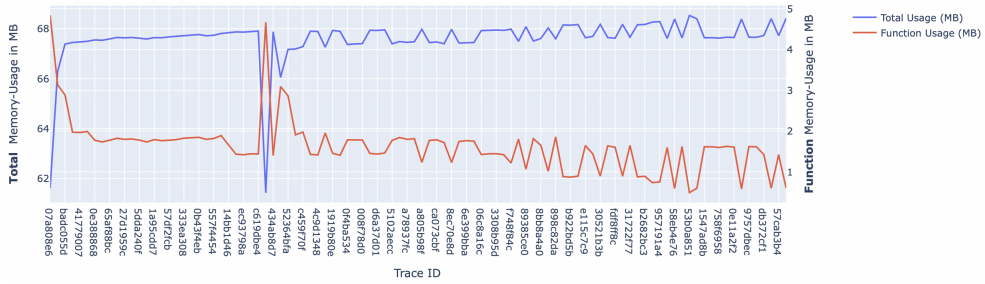
(a) Recorded average sent and received bytes.

(b) Recorded network connections.

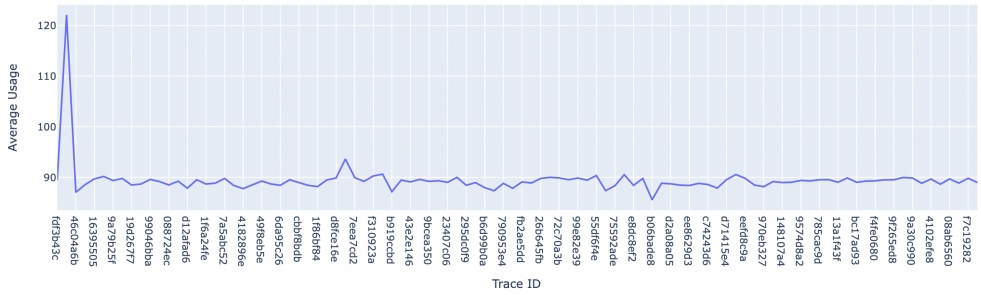
Figure 6.8: FaaS-Profiler visualization of network I/O and connection for *Website Download* on GCP Functions.

Website Download Figure 6.8 shows the results of the *Website Download* function executed on the Google Cloud Platform. The process was performed 50 times. Like Amazon, a stable network I/O rate can be measured, which is slightly about 0.5KB, lower than that on AWS Lambda. Running the I/O network counter measurement on the *No-op* showed that the empty function does not receive and send bytes. FaaS-Profiler records the more bytes consumed for the HTTP connection, even on Google Cloud. It is noticeable that Google uses an IPv6 connection by default, although the same source code was used for Lambda and GCP Function. The IPv6 address for `example.org` could be recorded in the majority of the calls.

Collecting the I/O counters and the network connections for GCP Function increased the average memory consumption from 82.12 MB to 127.16 MB by 54.85%. The execution time increased from 176.8 ms by 78.06% to 314.82 ms. The cost per GB-second increased by 175% on average.



(a) Recorded average memory consumption. Blue shows the total consumption and red the consumption only by the function itself. The sampling rate was $0.001s$



(b) Recorded average CPU consumption in percentage. The sampling rate was $0.1s$

Figure 6.9: FaaS-Profiler visualization of memory and CPU consumption for *Path Finding* on GCP Functions.

Path Finding Figure 6.9 shows the memory and CPU consumption of the Path Finding Method over 100 calls sorted by call time. Also, the red line shows the consumption of the function itself; previous consumption was subtracted. The blue line shows the total consumption. Only the consumption of the process in which the instrumented method is executed is monitored. You can also see a peak in the graph of memory consumption; this occurred at the time when a new instance was created.

Reading the CPU and memory consumption did not increase the execution time for Google Cloud Function. Memory consumption increased by 61.79%, from 87.19 MB to 141.0 MB. The cost per GB-second increased by 61.014% on average.

6.2 Applications

The following section presents three different serverless applications to which the FaaS-Profiler has been applied. The goal is to evaluate longer running applications and show how the tool reconstructs the temporal dependencies between function calls. We will first discuss the *Image Processing Pipeline* application that computes thumbnails of various images using asynchronous

calls. We then we discuss a longer running and more complex application that uses message queues and tasks to extract and store data from an API. In the last section, we present an implementation of a distributed matrix multiplication using AWS StepFunctions.

We will present visualizations of the workflow's FaaS-Profiler for each application. Here, the directed graphs are to be analyzed: The green round nodes symbolize serverless functions. These are labeled with the function key, the execution time in milliseconds, and the record ID. The size of a node indicates the execution time relative to the other nodes. An outgoing edge indicates that the function has made an outgoing request that FaaS-Profiler has recorded. The edge can point directly to another function or be connected to an intermediate service node. Service nodes are the gray rectangular nodes. These are labeled with the outgoing request type (service and operation) and the incoming request type. In the latter case, FaaS-Profiler recognized that another service is called another function. The thickness of the edge symbolizes the latency of the call. The thickness of an edge pointing to a service node indicates how long the function node took to upload a file or queue a task, for example. An outgoing edge from the service node indicates how long after the outgoing request is completed, another function is triggered[4].

The following section uses several cloud services integrated with serverless functions. We want to give a brief overview of them:

SQS [33, 22] is a distributed queuing service in AWS. It can send, store and receive messages and workloads between services. Messages can stay in SQS for up to 14 days; the default is four days. SQS is a polling-based service, though this is automatically taken care of when integrating with Lambda. Messages are not deleted until the consumer has successfully processed them. No messages are sent twice. The pooling mechanism usually results in higher latency, as notifications are stored and picked up later.

SNS [15, 22] is a distributed publish/subscribe service in AWS. A publisher sends messages to a topic, which many subscribers of that topic then consume. There can be many publishers and many subscribers. SNS generally has a short latency because messages are forwarded according to the fan-out principle. Messages are forwarded according to the push principle. SNS has no retention of messages; there is no guarantee that a message has been delivered. However, there are retry mechanisms in case the downstream service is unavailable.

[4]We assume that all the services' clocks are synchronized, which is not the case in reality. Therefore, the calculation is subject to uncertainty.

Cloud Tasks [46, 36] is a distributed task queue and, in many ways, comparable to Amazon's SQS. Cloud Tasks allows tasks to be queued and, unlike Pub/Sub, allows more control over the messages. Reception is guaranteed, and retry controls can be implemented. Compared to Pub/Sub, in Cloud Task, a function is called explicitly, while Pub/Sub functions are called implicitly.

Pub/Sub [45, 36] is Google's equivalent to SNS. It also works with the publisher-subscriber pattern based on topics. Publishers send messages to the service without knowing when and how the messages will be processed. Pub/Sub also generally has low latency, and all messages are forwarded to all subscribers according to the fan-out principle.

The default configuration was used for all services.

6.2.1 Image Processing

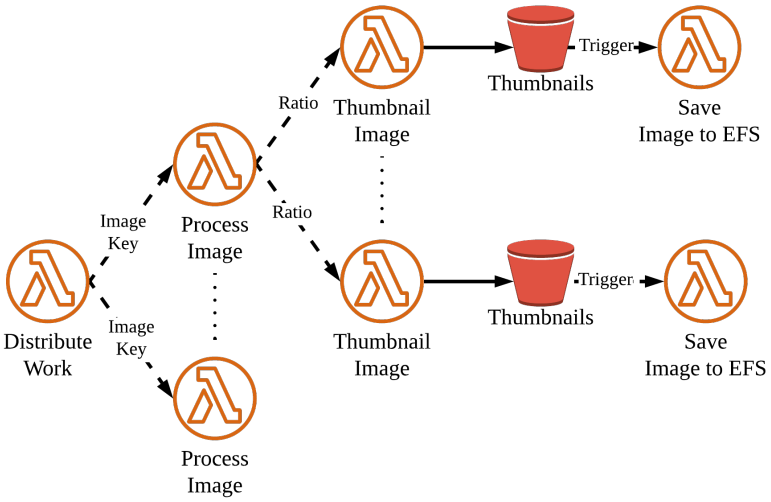


Figure 6.10: Structure of the image processing application

Figure 6.10 shows the schematic structure of the application for AWS. A folder in a bucket is passed to the *Distribute Work* function. A *Process Image* function is triggered asynchronously for each image in the folder. This triggers the *Thumbnail Image* function for various thumbnail ratios. *Thumbnail Image* calculates the image's thumbnail and stores it in a bucket. The *Save Image* function runs automatically when a new image is uploaded to this bucket. The function loads the image from the bucket and saves it as a backup in an EFS storage. This workflow was developed in parallel for Google Cloud Functions, omitting the last step due to the lack of EFS. To call the function asynchronously, we used a cloud task queue.

We run the application 20 times a row and let the FaaS-Profiler reconstruct the execution path.

AWS Lambda Results

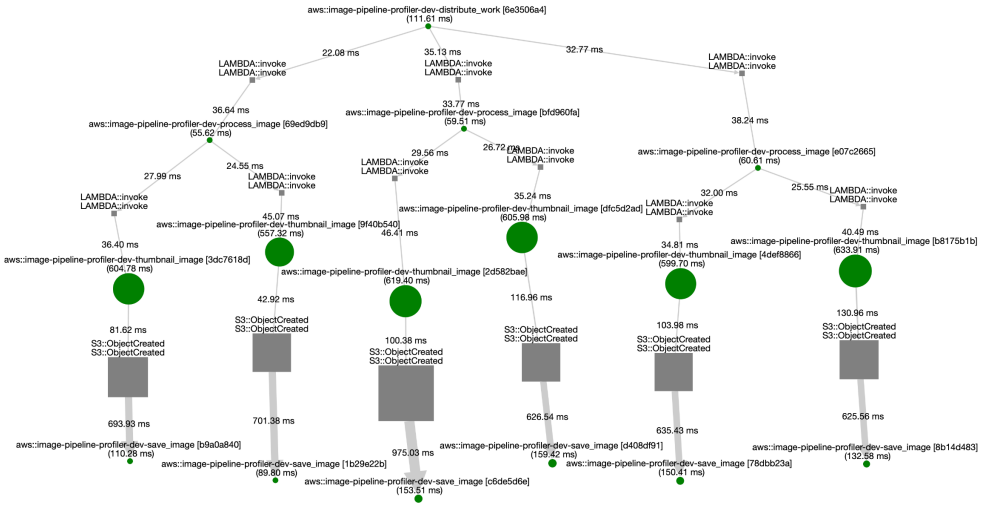


Figure 6.11: Visualization of FaaS-Profiler of *Distribute Work* for one execution reconstruction on AWS Lambda.

Figure 6.11 shows the reconstructed execution graph for AWS Lambda. In this case, *Distribute Work* commissioned three images to be processed. *Process Image* again ordered the image to be resized in ratios 5 and 10.

It can be seen that the trigger by S3 is significantly longer than the asynchronous call of a lambda function.

Overhead Recording this workflow has increased the average execution time of all functions. Distribute Work ran 41.2% (117.77ms to 166.37ms), Process Image 36.86% (65.78ms to 90.03ms), Thumbnail Image 11.83% (634.49ms to 709.57ms) and Save Image 3.95% (176.09ms to 183.03ms) slower. The memory consumption increased between 5.77% and 14.16% by a maximum of 11.1MB more.

Google Cloud Function Results

Figure 6.12 shows the reconstructed execution graph for Google Cloud Function. As with AWS Lambda, *Distribute Work* commissioned three images for processing, which were then ordered by *Process Image* to scale the image in ratios of 5 and 10. *Thumbnail Image* processes these images and stores the result in a bucket.

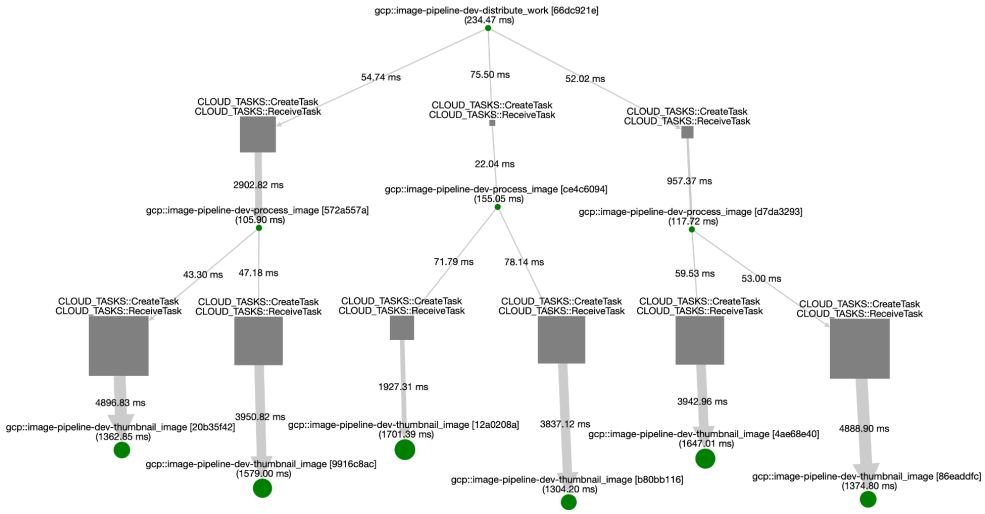


Figure 6.12: Visualization of FAAS-PROFILER of *Distribute Work* execution reconstruction on Google Cloud Platform.

It is noticeable that the latency of cloud tasks in Google Cloud is much higher than Lambda functions' asynchronous calls. The bottleneck in this workflow is the time spent to trigger all necessary functions. In Google, there is also the possibility to trigger a function directly through the Google Cloud Function API, but this is provided with a meager limit [42]. Based on the documentation, an asynchronous pattern must be implemented through queues [35].

Overhead The average execution time for tracing and exporting this application to Google Cloud increased from 7.56% to 43.18%. *Distribute Work* slowed down by 7.56%, from 209.5 ms to 225.35 ms. *Process Image* slowed 43.17% from 100.13 ms to 143.36 ms. *Thumbnail Image* time became 14.24% slower from 1519.06 ms to 1735.42 ms.

In terms of memory consumption, an increase of 8.45% to 10.10% on average was recorded. *Distribute Work* used 10.10% more memory from 112.78 MB previously to 124.18 MB. *Process Image* used 9.39% more memory from 111.15 MB to 121.59 MB. *Thumbnail Image* saw an 8.45% increase from 166.36 MB to 180.41 MB.

6.2.2 Quotes Event-Processing

Figure 6.13 shows the simplified schematic structure of the Quotes Applications. The application aims to retrieve quotes from a free API and store them in a database. We want to show the FAAS-PROFILER's ability to track function calls through event queues. The workflow was inspired and adapted from *Nietzsche* written by Abhishek Maharjan [53]. The entry point of the

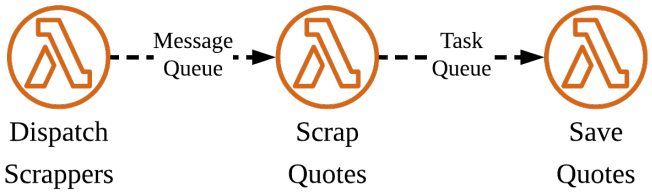


Figure 6.13: Structure of the Quotes application

application is the function *Dispatch Scrappers*. This contains ten different categories of citations. Each of these categories is published to a message queue. *Scrap Quotes* is triggered by one of the messages and asks a public API `api.quotable.io` to return all quotes of this category. The responses are packed into batches of size ten and published to a task queue. *Save Quotes* is triggered by one of these tasks and saves the batch to a database. Note that the API does not return quotes for every category. The goal was to show that FaaS-Profiler recognizes when an execution thread ends at *Scrap Quotes*. The application was tested on Amazon Web Services and Google Cloud Platform. For AWS, we took SNS and SQS as message and task queues, respectively. For GCP, we used Pub/Sub and Cloud Tasks. DynamoDB was used as the database in AWS and Datastore in GCP.

AWS Lambda Results

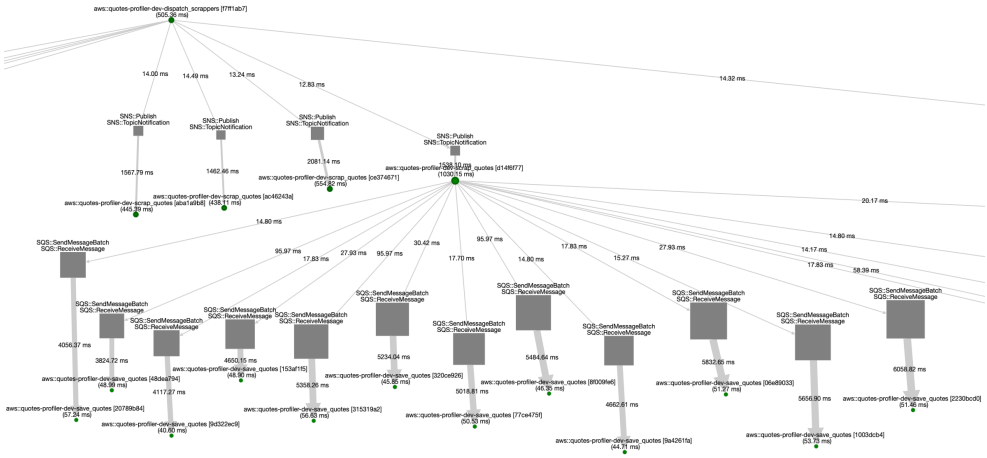


Figure 6.14: Part of the visualization of the FaaS-Profiler for the *Quotes* application on Amazon Web Services.

Figure 6.14 shows a portion of the reconstructed execution thread on Amazon Web Services. Based on the calculated latencies between the calls, it is

noticeable that SNS has less latency than SQS. As mentioned at the beginning, this difference is due to the way SNS and SQS work. While SNS passes messages directly with little latency, SQS has more overhead, for example, due to guaranteed delivery and storage of messages.

Overhead The recording and exporting of the executions increased the execution time by 21.95% to 53.70%. *Dispatch Scrappers* became 53.70% slower from an average of 176.19 ms to 270.81 ms. *Scrap Quotes* slowed down 23.94% from 306.59 ms to 379.99 ms; *Save Quotes* by 21.96% from 57.64 ms to 70.29 ms.

In terms of memory consumption, an increase from 19.94% to 22.05% was recorded. *Dispatch Scrappers* consumed 22.05% more memory from 70.3 MB before to 85.8 MB, *Scrap Quotes* consumed 22.04% more from 70.14 MB to 85.6 MB and *Save Quotes* consumption increased by 19.94% from 73.49 MB to 88.15 MB.

Google Cloud Functions Results

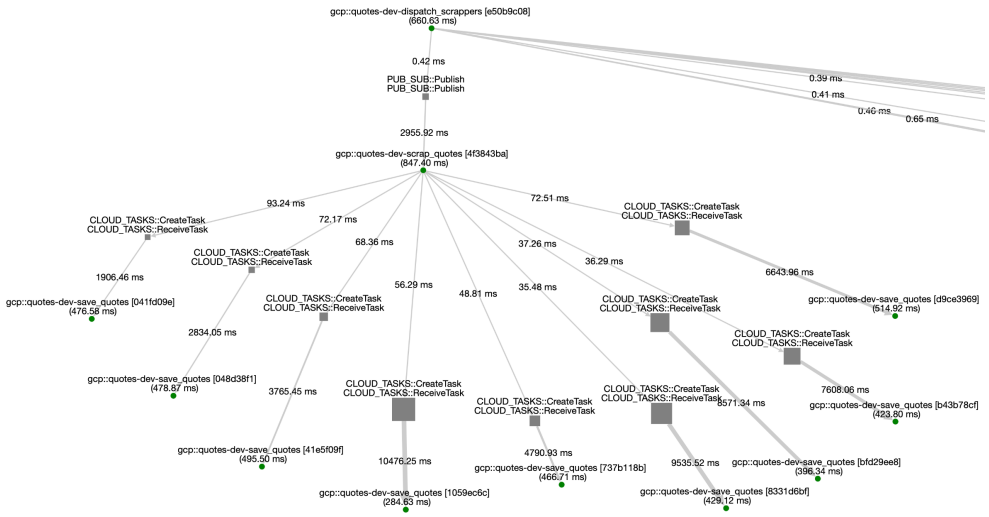


Figure 6.15: Part of the visualization of the FAAS-PROFILER for the *Quotes* application on Google Cloud Platform.

Figure 6.15 shows a partial visualization of the FAAS-PROFILER for the Quotes application performed on Google Cloud Platform. Similar to the execution on Amazon, one can see that the latency between calls by Pub/Sub is smaller than by Cloud tasks.

Overhead Overall, the average execution time increased between 8.31% and 57.89%. The execution time for *Dispatch Scrappers* increased by 8.31% from 356.6 ms to 386.2 ms. For *Scrap Quotes* it increased by 10.90% from 611.07 ms

to 677.70 ms. For *Save Quotes*, the time increased by 57.89% from 269.43 ms to 425.4 ms.

In terms of average memory consumption, we recorded an increase of 10.90% and 14.0%. The consumption for *Dispatch Scrappers* increased by 14.0% from 113.05 MB to 128.98 MB. For *Scrap Quotes* it increased by 10.90% from 108.51 MB to 120.34 MB. For *Save Quotes*, it increased by 12.54% from 105.19 MB to 118.39 MB.

6.2.3 Matrix Multiplication on AWS

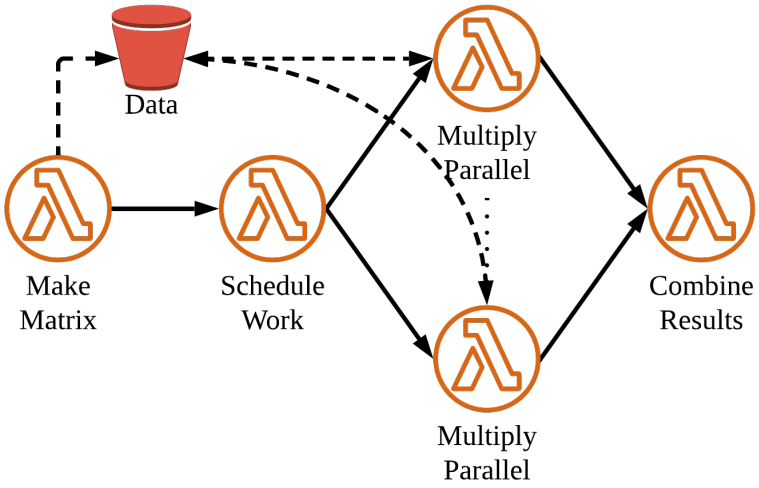


Figure 6.16: Structure of the matrix multiplication application

Matrix multiplication is a serverless function implemented using AWS Step-Function. Figure 6.16 shows the setup. Function *Make Matrix* creates two random matrices and stores them in a bucket. *Schedule Work* divides these matrices into submatrices based on the requested number of workers. *Multiply Parallel* loads the allocated submatrices, calculates the product, and stores the result in the bucket. *Combine Results* loads all sub-results into the result matrix.

Figure 6.17 shows the reconstructed graph of the FaaS-Profiler for matrix multiplication on AWS performed with step functions. Tracing was implemented by injecting the tracing context into the result of each step function. Note that the functions *Multiply Parallel* were declared parallel branches in AWS step functions, which do not return any results. For this reason, there is an edge from *Schedule Work* to *Combine Result* instead of edges from *Multiply Parallel* to *Combine Result*. It is also visible that the latency of this edge is visibly higher since *Combine Result* is called only after all *Multiply Parallel* nodes have completed. Tracing an application based on step functions has

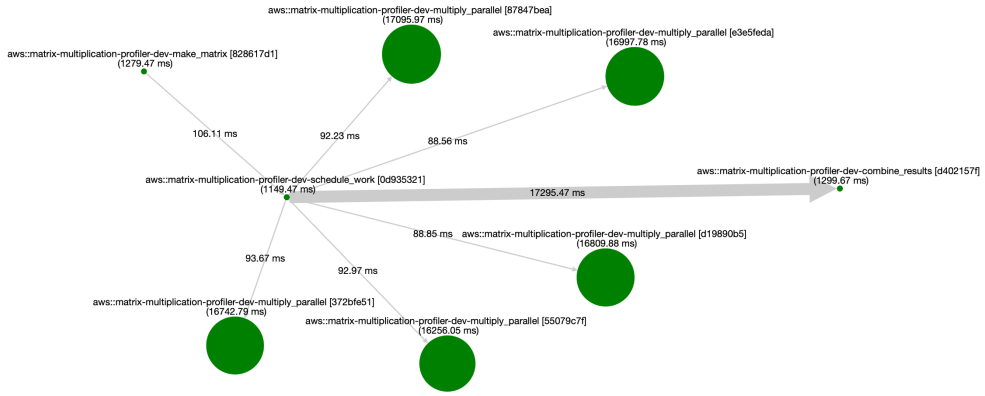


Figure 6.17: Visualization of FAAS-PROFILER of *Matrix Multiplication* workflow on AWS.

the difficulty that the functions do not know each other, and none of the functions make an outgoing request to be combined later. Therefore we chose the approach to inject the return values with the tracing context.

Chapter 7

Conclusion & Outlook

This thesis introduced FaaS-Profiler a framework to profile and trace serverless functions in the execution environment. We presented an open and easily extensible profiling format allowing serverless functions to be characterized, tracked, and populated with metrics independent of the programming language and FaaS platform. We also developed a system to perform measurements in the serverless execution environment to gain insight into a running function and to read performance metrics. This allows us to quantify code optimizations, and to identify bottlenecks.

FaaS-Profiler implements a tracer of serverless function calls to analyze dependencies and latencies between functions and services, to perform measurements across function boundaries, and to aggregate them. For this purpose, we introduce the notion of a request context, which generalizes incoming and outgoing requests of a function to a common denominator, to be able to make statements about this later and to reason about dependencies. We showed what information about incoming and outgoing requests needs to be collected to link them later temporally. Using the presented applications, we proved the ability of the distributed tracer to track and connect both synchronous and asynchronous invocations and triggers through services. We try to create a new approach for a basis to reconstruct execution graphs independent of the platform and the used event, which previous solutions could not do or were only applicable to one platform.

To prove the design, we implemented FaaS-Profiler in the form of a Python library with support for cloud providers Amazon Web Services and Google Cloud Platform. Following the high-level design presented, it provides all the functionality to port the library to another platform. With the implementation, we also showed which metrics can be provided in a serverless environment and presented a way to collect performance metrics via the virtual file system `procfs`. In addition to this process information, we showed a way to make performance statements about API calls to cloud services by presenting an

open and easy-to-use patching framework and function call documentation engine; this process has been seen in action with AWS S3 and AWS EFS.

In summary, FaaS-Profiler provides a foundation to use as an open framework to perform arbitrary measurements in the serverless environment while remaining platform independent. The tool first attempts to generalize serverless functions in a profiling format, making the function of different runtimes and providers comparable.

FaaS-Profiler can benchmark and experiment with serverless functionality in the research sector and compare these across platforms. We also see an application in production to detect bottlenecks and visualize the dependencies of a more extensive application. The open design also allows domain-specific measurement routines to be defined.

In the following, we discuss some limitations of the tool:

Scalability: The current implementation exports a record in JSON format for each run with a minimum size of about 1KB. This contains all data to trace and identify the function but no metrics. Currently, these are uploaded to cloud storage. The throughput of exporting to a bucket could become a bottleneck with many simultaneous calls. We have also developed a solution that uses DynamoDB as the export medium, which allows fast exporting, mainly because the records can already be partitioned by trace ID. In our experiments, however, using DynamoDB resulted in higher costs.

Dependency of request identifiers: The connection of function calls depends on the presence of request identifiers, which enforce the unique assignability of incoming and outgoing requests. This notion can be softened in further work to relax this strong assumption so that function calls with non-unique identifiers can still be connected.

Portability: As mentioned, many programming languages are supported in the serverless environment, while this work has focused only on Python. Any future library exporting the described format can use the FaaS-Profiler infrastructure with visualization and postprocessing algorithms. The limiting factor could be that in the programming language, patching is not possible, which could complicate collecting API metrics and intercepting outgoing requests.

In further work, additional *Instrumenters* can be written in other programming languages to make FaaS-Profiler applicable there as well. Also, we see not yet exhausted all possibilities to further reduce the overhead and performance for the tool.

Appendix A

Appendix

A.1 Assets

The Python implementation of the FaaS-Profiler can be found in the following repository:

github.com/spcl/faas-profiler-python

The visualizer and client line interface is located in the main repository:

github.com/spcl/faas-profiler

Furthermore, there is a helper repository with data structure definition and constants, which is used by the client and visualizer both:

github.com/spcl/faas-profiler-core

All applications presented in Chapter 6 as well as the result can be taken from the following repository for reproducibility:

github.com/spcl/faas-profiler/examples

A.2 Setup

This section gives practical advice on how serverless functions can be instrumented and analyzed with FaaS Profiler.

A.2.1 Client Deployment

For the profiler to collect data, the client must be installed in the serverless environment of the function. For Python, there is a package, `faas_profiler_python`, which can be installed via PIP and acts directly in the execution environment of the function.

AWS Lambda Amazon provides the ability to define layers that simplify the deployment of dependencies and reduce deployment time. For FaaS-Profiler, layers exist with pre-compiled code for x86_64 and is available under the Amazon Resouce Name:

```
arn:aws:lambda:eu-central-1:324305201550:layer:faas_profiler_python:71
```

Additional layers can be easily created using the Python `script/publish_layer.py` in the `faas-profiler-python` repository.

The client is automatically available for Lambda functions with added FaaS-Profiler-layer.

Google Cloud Functions Google Cloud installs and compiles all extensions automatically during deployment. For Python, the following dependency must be added to `requirements.txt`:

```
faas_profiler_python @
git+https://{GH_TOKEN}@github.com/spcl/faas-profiler-python
```

Where `GH_TOKEN` must be replaced with a valid GitHub key.

A.2.2 Instrumentation

To instrument any serverless function, the handler must be modified as follows:

```python
import faas_profiler_python as fp

@fp.profile(config_file = path_to_file | None)
def handler(*args, **kwargs):
    return {
        "message": "Hello FaaS-Profiler"
    }
```

A.2.3 Configuration

FaaS-Profiler offers the possibility to configure the framework via YAML or JSON files and environment variables. Via environment variables, we experienced the fastest setup time.

YAML/JSON Configuration

```yaml
profiler:
  measurement_interval: 0.1
  function_context:
    environment_variables: false
```

```
    response: false
    traceback: false
    payload: false

tracing:
  enabled: true
  inject_response: false
  trace_outgoing_requests:
  - gcp
  - aws

measurements:
  - name: cpu::UsageOverTime
  - name: cpu::Usage
  - name: cpu::UsageByCoresOverTime
  - name: cpu::UsageByCores
  - name: cpu::Usage
  - name: memory::Usage
  - name: network::Connections
  - name: network::IOCounters
  - name: information::Environment
  - name: information::OperatingSystem
  - name: information::IsWarm
  - name: information::TimeShift

captures:
  - name: aws::S3Access
  - name: aws::EFSAccess

exporters:
  - name: common::Console
  - name: storage::GCPVisualizerUploader
  - name: storage::AWSVisualizerUploader
```

As shown above, the format is divided into five sections. In the *Profiler* section, the global properties of the profiler can be defined. *Measurements*, *Exporters*, and *Captures* define a list of routines to be loaded and used in the respective category. Note that the client loads all routines lazy, i.e., the routine is imported only if requested. Each routine can be given a key-value mapping for optional parameters. In the section *Tracing*, the tracer is configured. It can be defined whether the response should be instrumented and which outgoing libraries should be traced.

Configuration via JSON follows the same format as YAML and is therefore omitted here.

A.2.4 Environment Variable Configuration

```
FP_MEASUREMENTS = "cpu::UsageOverTime,memory::Usage,..."
FP_CAPTURES = "aws::S3Access,aws::EFSAccess"
FP_EXPORTERS = "common::Console"

FP_PROCESS_INTERVAL = 0.1
FP_INCLUDE_VARS = False
FP_INCLUDE_RESPONSE = False
FP_INCLUDE_PAYLOAD = False

FP_ENABLE_TRACING = True
FP_TRACE_OUTGOING = "aws,gcp"
FP_INJECT_RESPONSE = False
```

The environment variables shown above are available for configuration. `FP_MEASUREMENTS`, `FP_CAPTURES`, and `FP_EXPORTERS` each define a comma-separated list of routines to be loaded and used.

A.3 Usage

To visualize and analyze the results of the FaaS-Profiler, the Python package `faas-profiler`, which can be installed via PIP, is used. After installation, commands are available to load the backtracing algorithm and the dashboard. It is assumed that the local user can access Amazon S3 and Google Cloud Storage, respectively.

To construct the trace graph the following command must be executed.

```
fp process_records \
   --provider PROVIDER \
   --region REGION \
   --project_id PROJECT_ID \
   --records_bucket BUCKET
```

`PROVIDER` must be replaced with either `gcp` or `aws`. `REGION`, `BUCKET`, and `PROJECT_ID` must be set so that read and write access to the cloud bucket in which the records were stored is possible.

To start the dashboard with all analyzers, the following command is used:

```
fp dashboard \
   --host "127.0.0.1"
   --provider PROVIDER \
   --region REGION \
   --project_id PROJECT_ID \
   --records_bucket BUCKET
```

Bibliography

[1] Performance application programming interface. https://icl.utk.edu/papi/. accessed: 14.09.2022.

[2] Jeff Barr. Aws lambda – run code in the cloud. https://aws.amazon.com/blogs/aws/run-code-cloud. accessed: 20.09.2022.

[3] Terrehon Bowden and Bodo Bauer. The /proc filesystem. https://www.kernel.org/doc/Documentation/filesystems/proc.txt. accessed: 16.09.2022.

[4] Mohak Chadha, Anshul Jindal, and Michael Gerndt. Architecture-specific performance optimization of compute-intensive faas functions. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 478–483. IEEE, 2021.

[5] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Graham Dumpleton. Wrapt - a python module for decorators, wrappers and monkey patching. https://github.com/GrahamDumpleton/wrapt. accessed: 16.09.2022.

[7] Giampaolo Rodola et al. psutil - cross-platform lib for process and system monitoring in python. https://github.com/giampaolo/psutil. accessed: 16.09.2022.

[8] Python Software Foundation. Python - built-in functions. https://docs.python.org/3/library/functions.html#open. accessed: 16.09.2022.

[9] Python Software Foundation. Python decorator definition. https://docs.python.org/3/glossary.html#term-decorator. accessed: 14.09.2022.

[10] Python Software Foundation. sys — system-specific parameters and functions. https://docs.python.org/3/library/sys.html#sys.settrace. accessed: 16.09.2022.

[11] Python Software Foundation. unittest.mock — mock object library. https://docs.python.org/3/library/unittest.mock.html, note=accessed: 12.09.2022.

[12] The Apache Software Foundation. Open source serverless cloud platform. https://openwhisk.apache.org. accessed: 12.09.2022.

[13] Preston Holmes and Google Inc. Asynchronous patterns for cloud functions. https://cloud.google.com/community/tutorials/cloud-functions-async. accessed: 22.09.2022.

[14] John Hunt. Monkey patching and attribute lookup. In *A Beginners Guide to Python 3 Programming*, pages 325–336. Springer, 2019.

[15] Amazon Web Services Inc. Amazon sns. https://aws.amazon.com/sns/. accessed: 04.05.2022.

[16] Amazon Web Services Inc. Aws cloudwatch. https://aws.amazon.com/cloudwatch. accessed: 20.09.2022.

[17] Amazon Web Services Inc. Aws lambda. https://aws.amazon.com/de/lambda. accessed: 12.09.2022.

[18] Amazon Web Services Inc. Aws lambda execution environment. https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html. accessed: 21.09.2022.

[19] Amazon Web Services Inc. Aws lambda runtime api. https://docs.aws.amazon.com/lambda/latest/dg/runtimes-api.html. accessed: 21.09.2022.

[20] Amazon Web Services Inc. Aws x-ray. https://aws.amazon.com/xray. accessed: 20.09.2022.

[21] Amazon Web Services Inc. Aws::lambda::function. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-lambda-function.html. accessed: 19.09.2022.

[22] Amazon Web Services Inc. Choosing between messaging services for serverless applications. https://aws.amazon.com/blogs/compute/choosing-between-messaging-services-for-serverless-applications. accessed: 04.05.2022.

[23] Amazon Web Services Inc. Comparing the effect of global scope. https://docs.aws.amazon.com/lambda/latest/operatorguide/global-scope.html. accessed: 18.09.2022.

[24] Amazon Web Services Inc. Deploy python lambda functions with .zip file archives. https://docs.aws.amazon.com/lambda/latest/dg/python-package.html. accessed: 22.09.2022.

[25] Amazon Web Services Inc. Deploying lambda functions. https://docs.aws.amazon.com/lambda/latest/dg/lambda-deploy-functions.html. accessed: 19.09.2022.

[26] Amazon Web Services Inc. Error handling and automatic retries in aws lambda. https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html. accessed: 22.09.2022.

[27] Amazon Web Services Inc. How do i give internet access to a lambda function that's connected to an amazon vpc? https://aws.amazon.com/premiumsupport/knowledge-center/internet-access-lambda-function/. accessed: 19.09.2022.

[28] Amazon Web Services Inc. Invoking lambda functions. https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html. accessed: 22.09.2022.

[29] Amazon Web Services Inc. Lambda runtimes. https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html. accessed: 22.09.2022.

[30] Amazon Web Services Inc. Using aws lambda with amazon eventbridge (cloudwatch events). https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html. accessed: 20.09.2022.

[31] Amazon Web Services Inc. Using aws lambda with aws x-ray. https://docs.aws.amazon.com/lambda/latest/dg/services-xray.html. accessed: 20.09.2022.

[32] Amazon Web Services Inc. Using aws lambda with other services. https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html. accessed: 22.09.2022.

[33] Amazon Web Services Inc. What is amazon simple queue service? https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html. accessed: 04.05.2022.

[34] Amazon Web Services Inc. What is aws step functions? https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html. accessed: 21.09.2022.

[35] Google Inc. Asynchronous patterns for cloud functions. https://cloud.google.com/community/tutorials/cloud-functions-async. accessed: 04.05.2022.

[36] Google Inc. Choose cloud tasks or pub/sub. https://cloud.google.com/tasks/docs/comp-pub-sub. accessed: 05.05.2022.

[37] Google Inc. Cloud functions. https://cloud.google.com/functions. accessed: 12.09.2022.

[38] Google Inc. Cloud functions execution environment. https://cloud.google.com/functions/docs/concepts/execution-environment. accessed: 19.09.2022.

[39] Google Inc. Cloudevents format - http protocol binding. https://cloud.google.com/eventarc/docs/cloudevents. accessed: 22.09.2022.

[40] Google Inc. Deploy a cloud function. https://cloud.google.com/functions/docs/deploy. accessed: 19.09.2022.

[41] Google Inc. Google cloud's operations suite. https://cloud.google.com/products/operations. accessed: 22.09.2022.

[42] Google Inc. Quotas. https://cloud.google.com/functions/quotas. accessed: 04.05.2022.

[43] Google Inc. Supported services. https://cloud.google.com/functions/docs/concepts/services. accessed: 22.09.2022.

[44] Google Inc. Understand workflows. https://cloud.google.com/workflows/docs/overview. accessed: 21.09.2022.

[45] Google Inc. What is pub/sub? https://cloud.google.com/pubsub/docs/overview. accessed: 22.09.2022.

[46] Google Inc. What is pub/sub? https://cloud.google.com/tasks. accessed: 04.05.2022.

[47] Google Inc. Write cloud functions. https://cloud.google.com/functions/docs/writing. accessed: 22.09.2022.

[48] Microsoft Inc. Azure functions. https://azure.microsoft.com/de-de/services/functions. accessed: 12.09.2022.

[49] Plotly Inc. Dash - analytical web apps for python, r, julia, and jupyter. https://github.com/plotly/dash. accessed: 16.09.2022.

[50] Michael Kerrisk. proc(5) — linux manual page. https://man7.org/linux/man-pages/man5/proc.5.html. accessed: 14.09.2022.

[51] Wei-Tsung Lin, Chandra Krintz, and Rich Wolski. Tracing function dependencies across clouds. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 253–260. IEEE, 2018.

[52] Wei-Tsung Lin, Chandra Krintz, Rich Wolski, Michael Zhang, Xiaogang Cai, Tongjun Li, and Weijin Xu. Tracking causal order in aws lambda applications. In *2018 IEEE international conference on cloud engineering (IC2E)*, pages 50–60. IEEE, 2018.

[53] Abhishek Maharjan. Nietzsche. https://github.com/rpidanny/Nietzsche. accessed: 04.05.2022.

[54] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE, 2017.

[55] Fabian Pedregosa and Philippe Gervais. Memory profiler. https://github.com/pythonprofilers/memory_profiler. accessed: 16.09.2022.

[56] Atsushi Sakai, Daniel Ingram, Joseph Dinius, Karan Chawla, Antonin Raffin, and Alexis Paques. Pythonrobotics: a python code collection of robotics algorithms. *arXiv preprint arXiv:1808.10703*, 2018.

[57] Joel Scheuner, Simon Eismann, Sacheendra Talluri, Erwin Van Eyk, Cristina Abad, Philipp Leitner, and Alexandru Iosup. Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications. *arXiv preprint arXiv:2205.07696*, 2022.

[58] Amazon Web Services. Boto3 documentation. https://boto3.amazonaws.com/v1/documentation/api/latest/index.html. accessed: 16.09.2022.

[59] Amazon Web Services. Botocore documentation. https://botocore.amazonaws.com/v1/documentation/api/latest/index.html. accessed: 16.09.2022.

[60] Amazon Web Services. Using amazon efs for aws lambda in your serverless applications. https://aws.amazon.com/blogs/compute/using-amazon-efs-for-aws-lambda-in-your-serverless-applications. accessed: 16.09.2022.

[61] Chris Siebenmann. Understanding resident set size and the rss problem on modern unixes. https://utcc.utoronto.ca/~cks/space/blog/unix/UnderstandingRSS. accessed: 16.09.2022.

[62] Vadim Struk. Introduction to serverless architecture in cloud-based applications. https://relevant.software/blog/serverless-architecture. accessed: 20.09.2022.

[63] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, 2021.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

FaaS-Profiler: Serverless Tracing and Profiling

Authored by (in block letters):

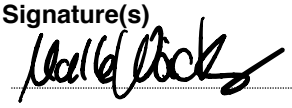
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Wächter	Malte

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 05 October 2022	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.